



National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

University of Alberta

WIRING OPTIMIZATION
IN
SLICING FLOORPLANS

by



Hossein Sahabi

A thesis
submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree
of Master of Science

Department of Electrical Engineering

Edmonton, Alberta

Fall, 1992



National Library
of Canada

Bibliothèque nationale
du Canada

Canadian Theses Service

Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-77127-5

Canada

UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: Hossein Sahabi

TITLE OF THESIS: Wiring Optimization In Slicing Floorplans

DEGREE FOR WHICH THIS THESIS WAS PRESENTED: Master of Science

YEAR THIS DEGREE GRANTED: 1992

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed)

Permanent Address:

#215, 11532 - 40 Avenue
Edmonton, Alberta
Canada

Dated: 22 July 1992

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Wiring Optimization In Slicing Floorplans** submitted by **Hossein Sahabi** in partial fulfillment of the requirements for the degree of **Master of Science**.

.....*Jack Mowchenko*.....

Supervisor: Dr. J. T. Mowchenko

.....*Brian Cockburn*.....

Dr. B. F. Cockburn

.....*E. S. Elmallah*.....

Dr. E. S. Elmallah

Date *July 20, 1992*

Abstract

Floorplanning is one of the first stages of the integrated circuit layout design process. During this step the geometry and location of modules on the chip surface is determined. The objectives which are typically considered in floorplanning are to optimize area, wiring, and performance of the chip. In this thesis, we are concerned with wiring optimization in a special class of floorplans called slicing floorplans. The problem is to adjust the module positions and orientations in a given slicing floorplan so as to minimize the total wire length. Two new algorithms are presented for this problem. The first algorithm is a branch and bound algorithm which searches for an optimum solution by considering all possible configurations of the given floorplan. The second one is a greedy heuristic algorithm which does not necessarily produce an optimum solution. The algorithms are evaluated by applying them to practical VLSI circuits and the results are encouraging. The branch and bound algorithm is only practical for small floorplans, while the heuristic algorithm is able to produce good solutions very quickly.

Acknowledgements

I would like to thank my supervisor, Dr. Jack Mowchenko, for his valuable guidance throughout this research. Partial support of Natural Sciences and Engineering Research Council and the Canadian Microelectronics Corporation are acknowledged.

As well, I would like to acknowledge the financial support provided by Iran Ministry of Culture and Higher Education throughout this research. I would like to thank my parents who were encouraging during all stages in my education, and I would like to thank my wife because of her patience and understanding in my university education.

Table of Contents

Chapter 1: Introduction	1
Chapter 2: Background	7
2.1. Floorplanning Formulation	7
2.2. Wire Length Estimation	9
2.3. Floorplanning Data Structures	11
2.4. Existing Floorplanning Algorithms	14
2.4.1. Stockmeyer's Algorithm	15
2.4.2. Force-Directed Relaxation Algorithms	19
2.4.3. The Wong and Liu Algorithm	22
2.4.4. Performance-Driven Layout	27
2.5. Conclusions and Discussion	31
Chapter 3: New Algorithms For Wiring Optimization	33
3.1. Problem Definition	33
3.2. An Overview of Problem Solutions	34
3.3. A New Branch and Bound Algorithm For Wiring Optimization	37
3.3.1. The Pre-Processing Algorithm	37
3.3.2. The Branch and Bound Algorithm	46
3.4. A Heuristic Algorithm For Wiring Optimization	55
3.5. Summary	59

Chapter 4: Implementation and Results	60
4.1. Preliminaries	60
4.2. Software Overview	62
4.2.1. Abstract Data Types	63
4.2.2. Input and Output Files	69
4.2.3. Graphic Representation of the Floorplans	71
4.3. Experimental Results	73
Chapter 5: Conclusions	85
References	89
Appendix	91

List of Tables

4.1. Statistics for the example circuits	73
4.2. The results of the branch and bound algorithm	74
4.3. The results of the branch and bound algorithm for 5% solutions	81
4.4. The results of the best-first algorithm with $k=1$	82
4.5. The results of the best-first algorithm with $k=2$	82
4.6. The results of the best-first algorithm with $k=3$	83
4.7. The results of the best-first algorithm with $k=5$	84

List of Figures

Body of Thesis

1.1. Slicing and non-slicing floorplans	4
1.2. Adjustments in slicing floorplans	5
2.1. Net bounding box and minimum spanning tree	10
2.2. A slicing floorplan with its polar graph representation	12
2.3. A slicing floorplan with its slicing tree representation	13
2.4. The shape function	16
2.5. Representation of a slicing floorplan with a normalized polish expression	25
2.6. Two combinational cells to illustrate the delay equation	30
3.1. The problem state space tree	35
3.2. Definition of $xmin()$	38
3.3. Calculation of $xmin()$ for an internal node	39
3.4. Calculation of $wmin()$ for an internal node	41
3.5. Updating $r()$ and $l()$	51
4.1. Description of .tre file	70
4.2. Apte module locations	75
4.3. Apte module locations and net bounding boxes	76
4.4. Plots of P_{best} changes in Apte	78
4.5. Plots of P_{best} changes in Hp	79
4.6. Plot of P_{low} in Apte	80

Appendix

A.1. Xerox module locations, produced by the branch and bound algorithm	91
A.2. Xerox module locations and net bounding boxes, produced by the branch and bound algorithm	92
A.3. Plots of P_{best} changes in Xerox.	93
A.4. Hp module locations, produced by the branch and bound algorithm	94
A.5. Hp module locations and net bounding boxes, produced by the branch and bound algorithm	95
A.6. Plot of P_{low} versus the number of times the $Update\_..()$ functions are called, in Xerox.	96
A.7. Plot of P_{low} versus the number of times the $Update\_..()$ functions are called, in Hp.	96
A.8. Ami33 module locations, produced by the best first algorithm with $k=5$	97
A.9. Ami33 module locations and net bounding boxes, produced by the best first algorithm with $k=5$	98

A.10. Ami49 module locations before running the algorithm	99
A.11. Ami49 module locations after running the best first algorithm with k=5	100
A.12. Ami49 module locations and net bounding boxes before running the algorithm	101
A.13. Ami49 module locations and net bounding boxes after running the best first algorithm with k=5	102

Chapter One

Introduction

The integrated circuit design process consists of two main activities. The first activity is design synthesis, which can be further broken down into behavioral or functional design, and physical or layout design. The second activity is design verification, which can be also further broken down into: logic verification, structural verification, and simulation. Various tools have been developed for each of these activities [16].

Because of the rapid growth in complexity, size and density of integrated circuits, layout design of VLSI systems is a challenging task. An efficient way to manage the complexity of the VLSI layout problem is through hierarchical design. With hierarchical design, the circuit is repeatedly partitioned into blocks, until each block contains a single circuit *module*. Each module has a number of I/O terminals on its boundaries. The circuit modules at the lowest level of the hierarchy are assumed to be indivisible circuits which are realized with standard cells or macro-cells. A standard cell implements a basic logic function such as a four input NAND gate, a D flipflop, *etc.* Macro-cells perform more complicated functions, such as an arithmetic logic unit (ALU), and their layout is produced either manually or with a module generator.

The hierarchical structure of a circuit layout can be represented by a tree. The root of the tree is the top level of the hierarchy and represents the whole circuit. Each internal node in the tree corresponds to a block which consists of a group of modules. An indivisible single circuit module is assigned to each leaf node at the lowest level of the tree; such modules are called *leaf cells*.

Even with computer assistance, manual layout design is a time consuming and error prone process; hence there is considerable interest in finding efficient ways to automate this activity. The dream of most computer-aided design tool developers has been to have the computer produce the layout for an entire integrated circuit without human intervention. Silicon compilation, which is the most recent approach to automated layout, comes close to this capability. Silicon compilers transform the structural description of a digital system into geometric mask information. The work done by a silicon compiler can be broken down into three major tasks: floorplanning, cell synthesis and chip assembly. Each of these tasks is discussed in the following paragraphs.

Floorplanning consists of arranging various circuit modules on the chip surface so as to satisfy a set of electrical and physical constraints, and to minimize a cost function which measures the quality of the layout. Typically, modules are assigned to rectangular regions and the leaf modules may have either fixed or flexible shapes. If all of the leaf modules have fixed shapes, the task of arranging modules on the chip surface is referred as macro-cell placement. Only when some of the modules have flexible shapes is the task referred to as floorplanning. Thus in floorplanning, in addition to deciding the location of modules, the aspect ratios of flexible shape modules must also be determined. Floorplanning is a difficult problem to solve, because of the many degrees of freedom in determining module locations and aspect ratios.

The second task performed by a silicon compiler is cell synthesis. This task consists of producing the mask layout for the circuit modules or leaf cells. There are two types of cell synthesizers. One type produces the layout only for a specific type of cell, such as an adder or a register. It takes as input some basic parameters describing the general characteristics of the cell, such as the number of

bits in a register, and produces a layout using a predefined template. The second type of cell synthesizer uses heuristic algorithms to produce the mask layout for any schematic, and the layout does not adhere to a predefined template.

The third task performed by a silicon compiler is chip assembly. This task consists of taking the floorplan and the leaf cell layouts and producing the layout for the entire chip. The main problem in this task is to determine the routing between the circuit module terminals. This may include shifting leaf cells in order to have enough space to accommodate wiring, if the required space is not available. Another operation which may occur in chip assembly is compaction of the layout to remove free space.

Among the three tasks which make up silicon compilation, this thesis is concerned with floorplanning. Different objectives are used by different floorplanning algorithms; these objectives are typically realized as factors in a cost function. We now review three important cost objectives.

One of the objectives that is considered in floorplanning is area utilization. Obviously, the layout which occupies a smaller area has a lower fabrication cost. The area used by a particular layout depends on the area of modules to be placed and on the area needed to complete the interconnections. Therefore, in order to minimize the chip area, the modules must be packed together optimally and the area occupied by interconnections must be reduced.

Another important objective that is addressed indirectly in chip area minimization, is total wire length minimization. Minimizing total wire length in a circuit can be carried out by placing the modules which are highly connected close to each other. Wire length minimization makes the routing task easier and less routing area is required, in general.

The third important objective in floorplanning which has recently received attention, is performance optimization. Of the various parameters which affect chip performance, interconnection delay is very important, since it directly reduces the speed of the circuit. By considering the effect of critical nets, modules which have a critical timing relationships can be placed such that the resulting floorplan gives the minimum path delay.

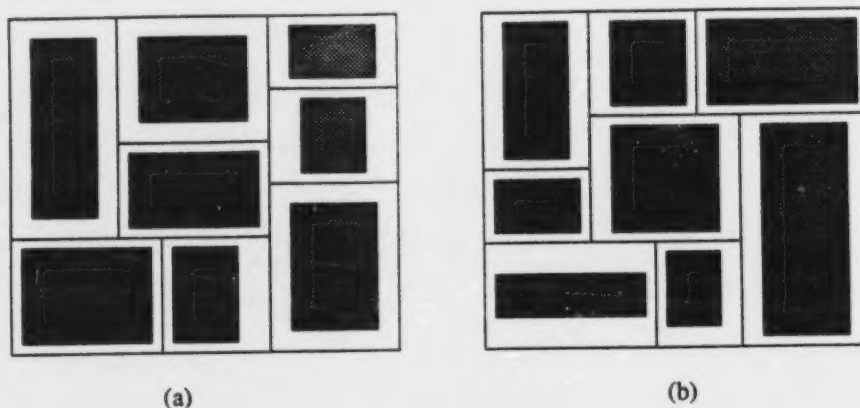


Figure 1-1: a) a slicing floorplan b) a non-slicing floorplan

After these discussions about various objectives in floorplanning, it is necessary to introduce an important class of floorplans. Given a floorplan, a layout can be enclosed by a rectangle so that it contains all the components of the layout. It is possible to subdivide this rectangle into a set of rectangles, each of which contains one and only one of the circuit modules. Each of these rectangles is called a *basic rectangle*. There are different ways to produce this rectangular decomposition. In some floorplans, the modules are arranged such that it is possible to produce the rectangular decomposition by recursively cutting the rectangles with either a vertical line or a horizontal line. Such floorplans are called *slicing floorplans* [17]. We are interested particularly in slicing floorplans for a number of

reasons which will be explained in the next chapter. If the recursive cutting method of partitioning is not feasible for a floorplan, it is called a non-slicing floorplan. Figure 1-1 illustrates a slicing and a non-slicing floorplan.

This thesis is concerned with wiring optimization in slicing floorplans. It is assumed that a slicing floorplan for a circuit already exists and the problem is to adjust the module positions and orientations so as to minimize the total wire length. In a slicing floorplan, each rectangle except the basic rectangles is split into two smaller subrectangles. The type of adjustments to module positions are restricted to interchanging the subrectangles produced by each slice line, and mirroring the modules in the basic rectangles. An example of these adjustments is illustrated in Figure 1-2.

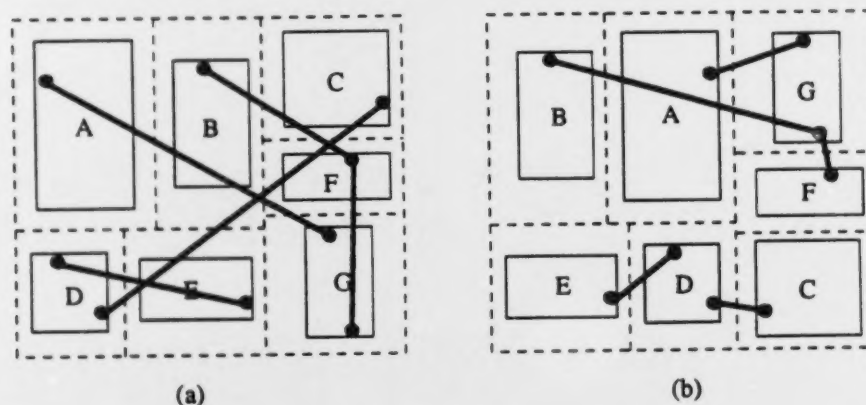


Figure 1-2: a) initial floorplan b) adjusted floorplan

In a circuit, terminals from different modules must be electrically connected to each other. A set of terminals which are to be interconnected is called a *net*. In Figure 1-2, ● represents module terminals, and diagonal lines symbolize the nets interconnecting the terminals.

As can be seen from Figure 1-2, the following adjustments have been made

to the initial floorplan:

- 1) The rectangle containing module A has been interchanged with the rectangle containing module B.
- 2) Module A has been mirrored about its vertical axis.
- 3) The rectangle containing module D has been interchanged with the rectangle containing module E.
- 4) The rectangle containing module G has been interchanged with the rectangle containing modules F and C. Then, the rectangle containing module F has been interchanged with the rectangle containing module C.
- 5) Module C has been mirrored about its vertical axis.

Figure 1-2 b) indicates that the adjustments bring terminals of some of the nets closer to each other, while terminals of some other nets may become farther from each other. Overall, the adjustments should tend to bring terminals of the nets closer to each other; this may reduce the total wire length of the circuit after completing the routing process.

This thesis explores different ways of solving the stated problem. In Chapter 2, a brief definition of the essential concepts of floorplanning is given, followed by some existing algorithms for the task. We mainly present the algorithms that take wiring optimization into account. In Chapter 3, the details of two new algorithms as solutions to our problem are presented. Chapter 4 describes the implementation of these algorithms and the results obtained by applying them to actual VLSI circuits. Conclusions and future work are given in Chapter 5.

Chapter Two

Background

This chapter is devoted to providing the necessary background definitions in the field of floorplanning, and to describing existing floorplanning algorithms. This chapter is organized in two sections. In section 1, the floorplanning problem is defined formally and methods to estimate the wire length in floorplans are described. As well, section 1 also describes the data structures used to represent a floorplan. In section 2, four floorplanning algorithms are described and evaluated.

2.1. Floorplanning Formulation

As was pointed out in the last chapter, floorplanning is one of the first stages of the IC layout design process. During this step the circuit modules are arranged on the chip surface. The geometries and locations of these modules must be selected such that the design quality, measured in terms of chip area, wiring density, power and timing considerations is optimized.

Floorplanning is a difficult problem because, in addition to the location of modules, the aspect ratio and I/O terminal positions on the boundary of modules must also be determined. Allowing a floorplanning algorithm to determine the aspect ratios and terminal positions is advantageous because the flexibility in deciding aspect ratios can be exploited to obtain better area utilization and performance. Furthermore, terminal positions on the boundary of the modules can be assigned such that the wiring area and the interconnection length are minimized.

Before discussing the various floorplanning algorithms, the problem must be clearly defined. The floorplanning problem can be formulated as follows:

A set of modules are given which have variable dimensions and I/O terminals with variable positions on the boundary of modules. The estimated delay and power consumption is available for each module. The circuit *net list*, which specifies terminals of the modules to be interconnected and estimated delay per unit length of interconnection, is also available. Finally, the following set of constraints are also given:

- Constraints on the location of some modules in the layout.

This restriction may take several forms. For example, the orientation of some of the modules may be fixed, the location of a component may be restricted to a specified region, the distance between two components may be bounded above or below by a certain limit, or two components may have to be placed symmetrically with respect to an axis of the chip.

- constraints on the overall chip area and aspect ratio.
- constraints on the aspect ratios of the modules.

Some of the modules in the design may have fixed dimensions; others may have aspect ratios that vary continuously or discretely.

- constraints on the position of I/O terminals on the boundary of modules.
- constraints on total estimated power dissipation.
- constraints on the estimated delay at the chip level.

The problem to be solved is to determine geometries, locations, aspect ratios and terminal positions for all the modules, so that all of the constraints are satisfied, and to minimize a cost function which is usually a weighted sum of total estimated area, wire length, delay and power dissipation.

2.2. Wire Length Estimation

The cost function which is to be optimized in floorplanning involves several components and varies depending on the nature of the design. Since area, total wire length and chip performance all depend on the results of the routing step, computing the exact value of the cost function requires that the layout be completely routed. However, it is absolutely impractical to route the layout when a floorplan is to be evaluated. Hence, being able to accurately estimate wiring quality is essential in chip floorplanning. The rationale behind our discussion about approximate wire length calculations in this section is that the total wire length (or total net length) is one of the most important components in the cost function and it also plays a key role in our proposed algorithms.

In VLSI circuits, net interconnections are usually implemented with rectilinear wires (manhattan geometries). Because of many different, but electrically equivalent, ways of interconnecting a set of terminals, even with manhattan wires, net length estimation is not an easy task. Heuristic methods which are fast to compute are required to solve the problem. Three popular methods for net length estimation are explained here:

1) Bounding box half perimeter

The half perimeter of the smallest rectangle enclosing all the terminals of the net has been used extensively to estimate the net length. For nets with manhattan geometries, this estimate is exact for two and three terminals nets. The total net length of the circuit is estimated with the sum of these half perimeters for all of the nets in the circuit. This estimate is easy and fast to compute. Figure 2-1 a) illustrates the bounding box for a net with five terminals.

2) Minimum spanning tree

A more realistic estimate of the net length is the length of the minimum spanning tree that connects the terminals of a net. Again, the total net length of the circuit is estimated by the sum of the lengths of these minimum spanning trees. Although there are algorithms to solve minimum spanning tree problem in polynomial time, its solution is much more time consuming than finding minimum bounding boxes and it is not practical for floorplanning algorithms that require frequent net length estimates. Figure 2-1 b) illustrates the manhattan minimum spanning tree of a net with five terminals.

3) Weighted distances between center of modules

In this approach, it is assumed that the I/O terminals of a module are all located in the center of module. The total net length of the circuit is estimated by the sum of the euclidean or manhattan distances between the centers of a pair of modules multiplied by the number of nets connecting the pair, over all pairs of modules in the circuit. If the modules are large in size, the actual position of the terminals may be far from the center of the module and this approximation is unrealistic. In spite of this problem, this estimate is easier to compute than the two preceding methods.

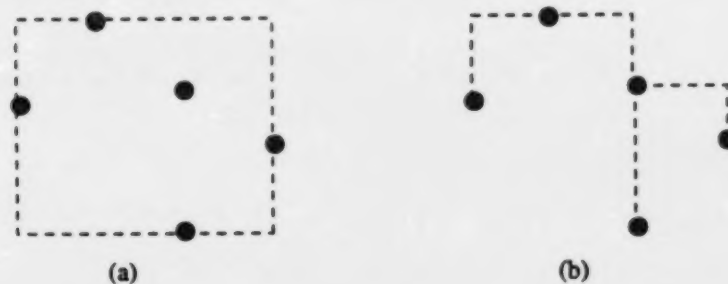


Figure 2-1: a) a bounding box b) a minimum spanning tree

2.3. Floorplanning Data Structures

In order to implement floorplanning algorithms, data structures are required to represent the relative positions of the floorplan components. In Chapter 1, it was mentioned that a floorplan can be subdivided into basic rectangles, each of which represents a region in the floorplan that contains only one circuit module. The rectangles which make up a floorplan are referred to as a *rectangle dissection*. The data structures which are used to represent a floorplan, represent this dissection. In this section we review two of these data structures: polar graphs and decomposition trees.

Polar Graphs

One way to represent a floorplan is by using two directed graphs called polar graphs. These graphs are known as *horizontal* and *vertical polar graphs*. The horizontal polar graph gives the relative position of the basic rectangles in the x coordinate. Each node in this graph corresponds to a vertical boundary of a basic rectangle. The nodes corresponding to left and right vertical boundaries of the entire floorplan are called the *source node* and the *sink node*, respectively. A directed edge between node i and j in this graph represents the fact that there exists a basic rectangle whose left and right boundaries are at the locations corresponding to node i and j , respectively. Hence, each of the edges in the graph corresponds to a basic rectangle in the floorplan; this is encoded in the graph by labeling each edge with its corresponding circuit module.

The vertical polar graph is the dual of the horizontal polar graph. This graph is constructed in the same way using horizontal lines instead of vertical lines in the floorplan. The source node in the graph corresponds to top-most horizontal boundary and the sink node corresponds to bottom-most horizontal boundary.

The two polar graphs can define a floorplan uniquely. If we label the edges in these graphs with the dimensions of the basic rectangles, then the longest path in the horizontal and vertical graphs gives the width and height of the overall floorplan.

The polar graphs that represent slicing floorplans have special characteristics. Each of these graphs can be reduced to a graph with only one source node and one sink node connected by a single edge. This can be carried out by a sequence of series and parallel reductions. Parallel reduction consists of replacing two or more edges that connect the same two nodes with a single edge, while series reduction consists of removing a node i from the graph with its incoming and outgoing edges (j,i) and (i,k) and replacing them with a single edge (j,k) . Because of this characteristic, the polar graphs representing slicing floorplans are called *series-parallel graphs*. Figure 2-2 illustrates a slicing floorplan with its corresponding polar graphs.

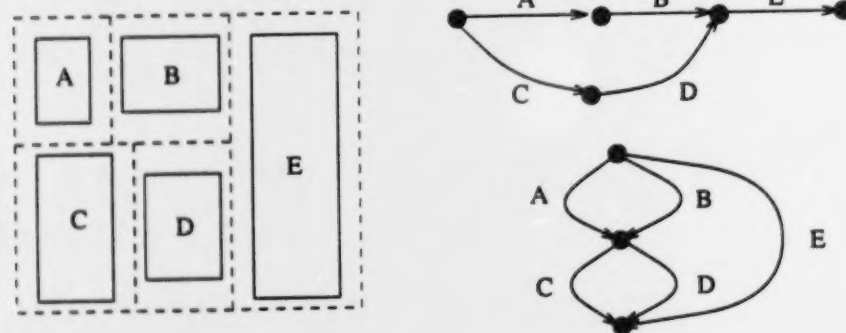


Figure 2-2: A slicing floorplan with its polar graph representation

Decomposition Trees

The other data structure which is extensively used to represent floorplans is a *decomposition tree*. The decomposition tree is a directed tree which represents the

hierarchy of rectangular regions in the layout. Each leaf in the tree symbolizes a basic rectangle in the floorplan and each internal node symbolizes a composite rectangle. In other words, each leaf node represents a circuit module and each internal node represents a group of modules. Although this data structure can be used for general floorplans, it is mainly used for slicing floorplans.

The decomposition tree which represents a slicing floorplan is called a *slicing tree*. A slicing tree is a binary tree in which each leaf node represents a basic rectangle and each internal node represents a composite rectangle in the floorplan. Each internal node contains the direction of the slice line which splits the composite rectangle represented by that node. Furthermore, there is an assumption about the relative position of a node's children in the tree. For example, it is assumed that if a node corresponds to a vertical slice, its left child represents a rectangle which is placed to the left of the rectangle represented by the right child. Similarly, if a node corresponds to a horizontal slice, its left child represents a rectangle which is below the rectangle represented by its right child.

One problem with slicing trees is that there can be many slicing trees for each slicing floorplan. Figure 2-3 illustrates a slicing floorplan and one of its possible slicing trees.

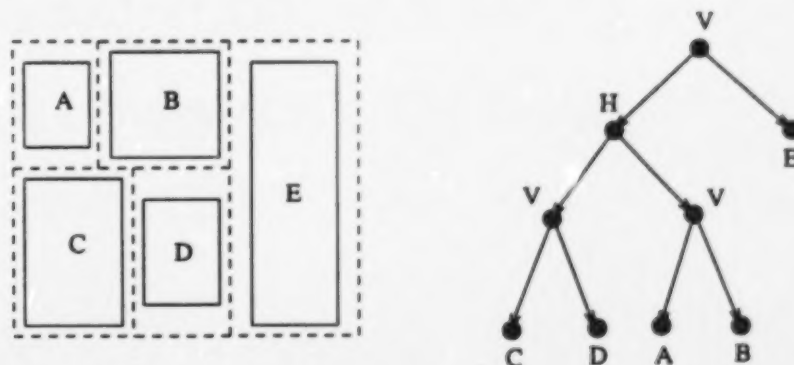


Figure 2-3: A slicing floorplan with its slicing tree representation

2.4. Existing Floorplanning Algorithms

Some floorplanning algorithms deal specifically with the slicing floorplans. Slicing floorplans are distinguished by a number of interesting features as follows:

- 1) A slicing floorplan can be represented by a series-parallel graph or a binary tree (slicing tree).
- 2) They are ideally suited for hierarchical layout.
- 3) The routing task for slicing floorplans can be completely carried out by channel routers.¹
- 4) For a given slicing floorplan, there is an efficient algorithm to determine optimal shapes and orientations of modules. Conversely, it has been proved that orientation optimization for more general layouts is NP-complete.
- 5) There are efficient algorithms that can produce slicing floorplans.

Because of the complexity of floorplanning problem, the existing algorithms only attempt to optimize a subset of all the different quality measures. In this section four such algorithms will be described. The first algorithm was developed by Stockmeyer [12], and it is the one mentioned above which is able to determine the optimal shape and orientation of modules for a given slicing floorplan in polynomial time. The second algorithm is from a class of algorithms called "force-directed algorithms" which represent the interconnection between modules with a set of forces. The objective of these algorithms is usually wiring optimization, and they do not necessarily produce slicing floorplans. The third algorithm was described by Wong and Liu [15] and uses a method called *simulated annealing* to generate optimal slicing floorplans. The objective of this algorithm is a

¹ The routing regions in the layout are decomposed into rectangular regions that contain terminals on two sides only, called *channels*. The routing task is usually carried out in two stages: global routing and detailed routing. A channel router is an effective detailed routing tool for routing of channels.

combination of area and total wire length minimization. Jackson and Kuh introduced the fourth algorithm [3], which is concerned with chip performance optimization. This algorithm combines timing analysis and physical design to dynamically optimize the performance of the chip during floorplanning.

2.4.1. Stockmeyer's Algorithm

In this algorithm, it is assumed a slicing floorplan described by a slicing tree is given. Each leaf cell in the circuit has two dimensions, a and b , and it has two possible orientations depending on whether side a or side b is horizontal. The algorithm is also given a cost function $\psi(h,w)$, which is nondecreasing in both arguments h and w and computable in constant time. The cost function is an arbitrary function that could measure area or perimeter, for example, and the two arguments h and w are the height and width of the entire floorplan. The objective of the algorithm is to find the orientations of all the modules in the circuit such that the cost function is minimized.

The algorithm constructs a *shape function* for all rectangular regions in the floorplan which describes all possible dimensions for the rectangular regions. An instance of the shape function for a basic rectangle corresponding to a leaf cell in the floorplan with dimensions a and b is illustrated in Figure 2-4. If w and h are the width and height of the basic rectangle, the following conditions must be satisfied so that the basic rectangle can accommodate the leaf cell:

$$w > a, h > b \quad \text{or} \quad w > b, h > a$$

In Figure 2-4, the shaded area represents the set of all pairs (w,h) that satisfy the above conditions. Stockmeyer's algorithm describes a shape function with a set of sorted ordered pairs which give the minimum possible dimensions as follows:

$$\{(h_1, w_1), (h_2, w_2), \dots, (h_k, w_k)\} \quad h_i > h_{i+1}, w_i < w_{i+1} \text{ for } 1 \leq i \leq k$$

In Figure 2-4, assuming $a > b$, the shape function is described as follows:

$$\{(a, b), (b, a)\}$$

If $a = b$ or the leaf cell has a fixed orientation, the shape function is different and the above set consists of only one pair.

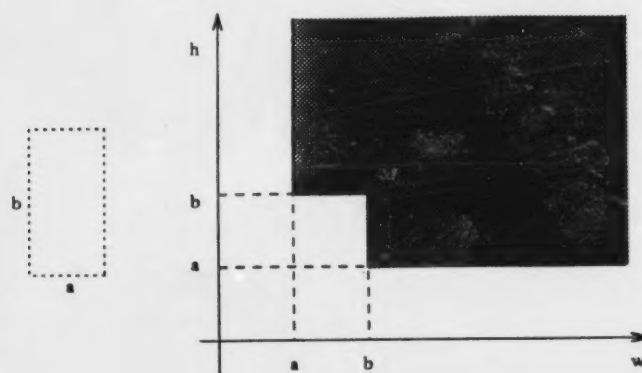


Figure 2-4: The shape function

Stockmeyer's algorithm works by moving bottom up through the tree. For each internal node u , the shape function of its children v and v' are combined to construct the shape function for u . Let us assume node u corresponds to a vertical slice in the floorplan and the shape function associated with v and v' are as follows:

$$\begin{array}{ll} \{(h_1, w_1), (h_2, w_2), \dots, (h_k, w_k)\} & h_i > h_{i+1}, w_i < w_{i+1} \text{ for } 1 \leq i \leq k \\ \{(h'_1, w'_1), (h'_2, w'_2), \dots, (h'_m, w'_m)\} & h'_i > h'_{i+1}, w'_i < w'_{i+1} \text{ for } 1 \leq i \leq m \end{array}$$

The combination procedure pseudo-code is as follows:

```

Combine(Shape(v),Shape(v'))
{
  i=1; j=1;
  Shape(u) = {};
loop:  while(i≤k and j≤m)
      {
        Add join((hi,wi),(h'j,w'j)) to Shape(u);
        if(hi>h'j){
          i=i+1;
          goto loop;
        }
        if(hi<h'j){
          j=j+1;
          goto loop;
        }
        if(hi=h'j){
          i=i+1;
          j=j+1;
        }
      }
}

```

In this procedure the join function is defined as follows:

$$\text{join}((h_i, w_i), (h'_j, w'_j)) = (\max(h_i, h'_j), w_i + w'_j)$$

The important point in this procedure is that all k pairs in the shape function of node v will not be combined with all m pairs in the shape function of node v' to produce $k \times m$ pairs in the shape function of node u . To illustrate this, consider the case where u corresponds to a vertical slice. If two pairs (h_i, w_i) and (h'_j, w'_j) are combined and added to the shape function of node u , and $h_i > h'_j$, it is not necessary to consider combining of (h_i, w_i) with (h'_k, w'_k) for any $k > j$. The combination of (h_i, w_i) with (h'_j, w'_j) is:

$$\text{join}((h_i, w_i), (h'_j, w'_j)) = (\max(h_i, h'_j), w_i + w'_j)$$

and the combination of (h_i, w_i) with (h'_k, w'_k) is:

$$\text{join}((h_i, w_i), (h'_k, w'_k)) = (\max(h_i, h'_k), w_i + w'_k)$$

and since the pairs in the $\text{Shape}(v)$ and $\text{Shape}(v')$ are sorted in decreasing order

of h and increasing order of w , we have:

$$\begin{aligned} \max(h_i, h'_j) = \max(h_i, h'_k) = h_i \\ w_i + w'_k > w_i + w'_j \end{aligned}$$

Thus, $\text{join}((h_i, w_i), (h'_k, w'_k))$ has the same first element as $\text{join}((h_i, w_i), (h'_j, w'_j))$, but a larger second element, and therefore should not be added to the shape function for node u .

The construction of shape functions for internal nodes is continued until the shape function for the root is constructed. From this shape function, the pair which minimizes the cost function can be selected and the cell orientations corresponding to this pair can be re-constructed. To aid in re-constructing the cell orientations, the algorithm keeps two pointers for each pair in the shape function. These pointers point to the pairs that combined to produce the given pair. The pointers in the shape function of basic rectangles are null. By following these pointers, the algorithm is able to find the optimum orientation for each leaf cell.

Stockmeyer's algorithm has been generalized by Otten [8] for determining the optimal shape of leaf cells with several different possible implementations. Otten's algorithm is basically the same as Stockmeyer's; however, the shape function for basic rectangles contains more than two pairs.

Stockmeyer's algorithm is used in some floorplanning algorithms to compute the optimal shape of the flexible modules in a given slicing floorplan [6]. The running time and storage requirements of Stockmeyer's algorithm are both $O(nd)$, where n is the number of leaf cells and d is the depth of slicing tree. The worst case complexity of this algorithm is $O(n^2)$.

2.4.2. Force-Directed Relaxation Algorithms

The basic idea behind this class of algorithms is to represent the interconnections between modules with a set of forces which are proportional to the number of interconnections and the distances between the modules. The objective is to find an arrangement such that the sum of all the forces acting on the modules is minimum. In this situation, each net's terminal will be closer to one another and this tends to minimize total wire length. Force directed algorithms are usually applied to a floorplanning problem in two phases. First, the algorithm is employed to obtain an initial solution. This phase is referred as the *constructive phase*. Second, the initial solution is modified incrementally to improve the quality of solution. This phase is called *iterative improvement*. Clearly, these algorithms do not necessarily produce slicing floorplans. Force directed algorithms have found wide application in a number of VLSI layout systems such as the PLINT system developed by General Electric for standard and macro-cells [1]. In this section the details of the force-directed relaxation algorithm implemented in a package called *CHAMP* is discussed [13].

In *CHAMP*, the floorplan is represented by an *interconnection graph*. Each node in this graph represents a module in the floorplan and the interconnections between modules are represented by edges, where each edge is given a weight proportional to the number of interconnections.

For the interconnection graph, a matrix E is defined with elements as follows:

$$E_{ij} = \begin{cases} \min(P_{ij}) & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases}$$

where $\{P_{ij}\}$ is the set of all distances along different paths between two nodes i and j . For each path, the distance is defined as the sum of the reciprocal of the

weight of each edge between two nodes for all the nodes in the path.

From E a new matrix R , called the *relativity matrix*, is defined with elements as follows:

$$R_{ij} = \begin{cases} E_{ij} - A_v & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases}$$

where A_v is the average value over all E_{ij} 's for $i \neq j$. From this matrix two sets of nodes will be defined for each node i . A strongly connected node set F_i , is the set of all nodes j with negative value of R_{ij} , and the weakly connected node set U_i , is the set of all nodes j with positive value of R_{ij} . It is assumed a set of attractive forces proportional to $|R_{ij}|$ work on node i from the nodes belonging to F_i ; alternatively, a set of repulsive forces act on node i from the nodes belonging to U_i .

In the constructive phase, each module is modeled by a node and there is no shape consideration. All the nodes are labeled by module id-numbers, and they are processed in the order of their id-numbers. First, node i is moved to the gravitational center of nodes belonging to F_i . Then node i is moved under the influence of repulsive forces of nodes belonging to U_i , so that the minimum force works on it. This procedure is repeated for all of the nodes in the graph. After several cycles of this process, the node placement is expected to converge into a certain arrangement which becomes the placement produced by the constructive phase.

After an initial placement is obtained, each module is given a rectangular shape and an appropriate chip boundary is set up enclosing all of the modules. The overlaps between modules or modules and chip boundary are eliminated during the iterative improvement phase by moving modules and/or by reshaping modules with the chip boundary being shrunk. The movements or reshaping ratios

are so small that they do not dramatically change the structure of the floorplan. This procedure has four steps as follows:

- 1) The chip boundary is shrunk by a given small value.
- 2) One module is selected arbitrarily, moved temporarily in four directions by a given small distance, and the amount of overlap is calculated for the other blocks as well as the chip boundary. Then the module is actually moved in the direction which leads to the maximum reduction in overlap.
- 3) One module is selected arbitrarily and its aspect ratio is temporarily modified by a given set of small values. Then it is actually modified to the aspect ratio which leads to the maximum reduction in overlapping areas.
- 4) Iterate step 1 to 3 until a satisfactory result is achieved. To judge the quality of floorplan, the chip area, including the wiring area, is estimated.

Experimental results demonstrate that the iterative improvement phase is the most time consuming part of this particular algorithm. This procedure consists mainly of selecting pairs of modules and calculating overlapping areas between them. If all pairs of modules are tried in the calculating process, the algorithm is $O(n^2)$. However, if only neighboring modules are considered for overlapping calculations, the algorithm becomes $O(n)$.

In general, the solution produced by the constructive phase of this algorithm is not satisfactory because of the greedy nature of the algorithm and the fact that it does not consider the module shapes. Therefore, the iterative improvement phase is essential to achieve acceptable solutions. Nevertheless, the final result in some cases is not satisfactory as too much unused space is left. This is due to the fact that this algorithm aims to optimize total wire length and does not directly address chip area.

2.4.3. The Wong and Liu Algorithm

This algorithm employs the technique of *simulated annealing* to solve the floorplanning problem, and simultaneously takes into consideration chip area and total wire length. First, the simulated annealing algorithm is discussed.

Simulated annealing was proposed by Kirkpatrick *et al.* [5], as an efficient method for determining global minimas of combinatorial optimization problems involving many degrees of freedom. In general, a combinatorial optimization problem consists of finding the minimum or maximum value of a cost function which depends on many independent variables. This problem arises in many different fields of science and engineering. Basically, simulated annealing is a *Monte Carlo* technique which simulates the equilibrium states of a collection of atoms at any given temperature T . According to Boltzman's law, the probability of existence of any configuration i of atoms is given by $e^{-E(i)/K_b T}$, where $E(i)$ is the energy associated with the configuration, and K_b is the Boltzman's constant. Hence, the most probable configurations at any temperature are those with the lowest energy.

In order to solve a combinatorial optimization problem, the cost function is used in place of energy. In the physical annealing process, the physical system moves from one system configuration to another. Correspondingly, the simulated annealing algorithm moves from one potential problem solution to another. The rate at which simulated annealing moves to a new solution is determined by the so-called temperature parameter T , just as the rate of change of atomic configurations in physical annealing depends on actual temperature. The simulated annealing process consists of first "melting" the system being optimized at a high effective temperature, and then lowering the temperature in small amounts based on a temperature "cooling" schedule, until the system "freezes" and no further

changes occur. A new configuration of the system is accepted unconditionally if its cost value is less than the cost of the previous configuration. It is accepted with probability $e^{-\Delta c/T}$ if the cost value associated with the new configuration is greater than with the previous configuration by Δc . A basic feature of the method is that the algorithm tends to escape from local minima because a transition out of a local minima is always possible at nonzero temperatures.

The basic structure of the simulated annealing algorithm is presented by the following pseudo-code:

```

Simulated Annealing Algorithm ( $j_0, T_0$ )
{
    /*  $j_0$  is the given initial state */
    /*  $T_0$  is the initial value for the parameter  $T$  */
     $T = T_0$ ;
     $X = j_0$ ;

    while( stopping criterion is not satisfied)
    {
        while( inner loop criterion is not satisfied)
        {
             $j = \text{generate}(X)$  ;

            if( accept ( $c(j), c(X), T$ )
             $X = j$ ;
            }
             $T = \text{update}(T)$ ;
        }
    }
}

```

In the above pseudo-code, T is the temperature, and X and j are the problem states. Each configuration of the problem which is a potential solution, is referred as a *problem state*. In this algorithm, the acceptance of a new solution is determined by the *accept()* function coded as follows, where j and i are the problem states, $f()$ is a function which takes values on the interval $(0,1]$, and *random*(0,1) is a function which returns a random number between 0 and 1.


```

accept (c(j),c(i),T)
{
     $\Delta c = c(j) - c(i);$ 
     $y = f(\Delta c, T);$ 

    /* In the original simulated annealing algorithm we have */
    /*  $f(\Delta c, T) = \min(1, e^{-\Delta c/T})$  */

     $r = \text{random}(0,1) ;$ 
    if( $r \leq y$ )
        return 1;
    else
        return 0;
}

```

The simulated annealing method is applied to the standard cell placement problem [5], macro-cell placement [4] and global wiring [14]. In this section, we discuss about the Wong and Liu algorithm [15] which uses simulated annealing to generate an optimal slicing floorplan. The basic idea behind this approach is to start with an initial slicing floorplan which could be obtained by some other algorithm, and then move from the initial solution to new solutions in search of the slicing floorplan that minimizes the total wire length as well as the total area of the chip.

In order to apply simulated annealing to the floorplanning problem, the cost function, the *generate* and *update* procedures, and also the *inner loop* and *stopping criteria* must be clearly determined. Wong and Liu applied the cost function defined as follows:

$$\psi = A + \lambda_w \times W$$

where A is the chip area, W is the total wire length of the layout using the weighted distances between center of modules, and λ_w is a weighting factor which determines the relative importance of the previous two parameters.

The *generate* procedure proposed by Wong and Liu is based on the formal

representation of slicing floorplans with *normalized polish expressions*. A normalized polish expression $\alpha_1\alpha_2\alpha_3 \cdots \alpha_{2n-1}$ has the following properties:

- 1) α_i belongs to the set $\{1, 2, \dots, n, *, +\}$ for $1 \leq i \leq 2n-1$. Numbers in the sequence are called *operands*, $+$ and $*$ are called *operators*
- 2) Each number from the above set appears only once in the sequence.
- 3) The total number of operators must be less than the total number of operands in the sequence.
- 4) There are no consecutive operators of the same type in the sequence.

It has been proved that there is a one-to-one correspondence between the set of normalized polish expressions of length $2n-1$, and the set of slicing floorplans with n basic rectangles, provided that each number in the sequence identifies one of the leaf cells and $*$ and $+$ represent horizontal and vertical slices, respectively. Figure 2-5 illustrates a slicing floorplan along with its corresponding normalized polish expression.

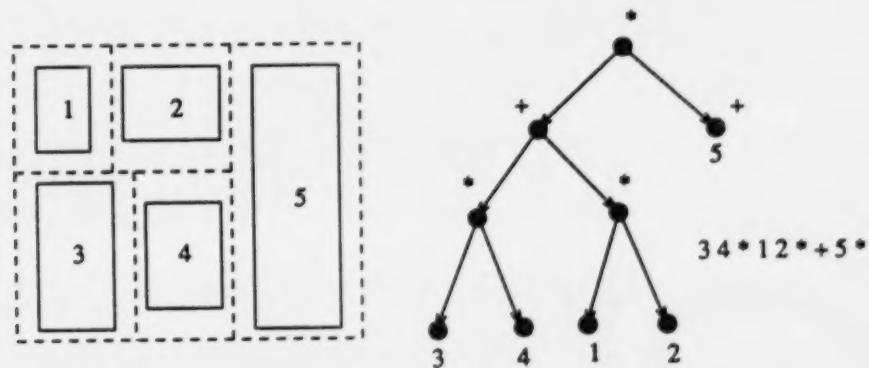


Figure 2-5: Representation of a slicing floorplan with a normalized polish expression

Each normalized polish expression can be modified to produce another normalized polish expression with one of the three following moves:

- 1) Swap two adjacent operands.
- 2) Interchanging all *'s and +'s in the sequence.
- 3) Swap an adjacent operand and operator.

These three moves are sufficient to ensure that it is possible to go from one normalized polish expression to any other via a sequence of moves.

The *generate* procedure consists of applying randomly one of these three moves to the present configuration of the system. The initial state of the system consists of placing the n modules next to each other horizontally, corresponding to the normalized polish expression $12*3*4...*n$. This initial state is usually far from the optimal solution.

The *update* procedure in the Wong and Liu algorithm is a fixed ratio temperature schedule.

$$T_k = r \times T_{k-1}$$

In this equation, r is an important constant which must be set to a suitable value in order to achieve satisfactory results.

The inner loop criterion is a condition which must be met to terminate the procedure for a particular temperature. This criterion is satisfied when the number of downhill moves exceeds N or the total number of moves exceeds $2N$ where N is $O(n)$. A downhill move is a move which leads to a configuration of the system with lower cost. The algorithm stops when the temperature falls below a limit or the number of accepted moves is less than 5% of all moves made at a certain temperature.

Although the Wong and Liu algorithm is elegant and simple, it is restricted to slicing floorplans. As well, non-rectangular modules and variable terminal positions can not be handled. Finally, the total wire length estimation in this algorithm is rather poor, since terminal positions are only approximated by the centers

of modules.

Because the Wong and Liu algorithm is based on the simulated annealing, it is interesting to mention another attempt to solve the floorplanning problem using this approach. The TimberWolf package [11] is a more sophisticated implementation of simulated annealing to solve the floorplanning problem which includes the following basic features:

- 1) Cells may be represented by any manhattan polygon.
- 2) Cells have aspect ratios that vary continuously or discretely over a range.
- 3) Cells may have variable terminal positions.
- 4) Weights may be assigned to each net to bias the placement.

As with the Wong and Liu algorithm, the objective of TimberWolf is total wire length and area optimization.

2.4.4. Performance-Driven Layout

The previous algorithms reviewed in this chapter are aimed at attaining chip area and/or total wire length minimization during floorplanning. However, wire length or area minimization as a sole objective does not necessarily improve chip performance.

From various aspects of chip performance, timing has an important role. In order to improve the chip timing performance, the interconnection delays must be reduced by considering timing constraints during floorplanning. Specifically, the effect of critical delay paths in the circuit must be considered and the modules that have restricted timing relationships must be placed such that minimum path delay is obtained.

Several approaches have been proposed for optimizing chip performance during floorplanning. One common method is net weighting [10]. This involves

transferring timing constraints on critical nets to weight factors assigned to them, and finding an arrangement of modules which has a minimum sum of weighted net lengths. This technique is repeated and weights are adjusted until all timing constraints are met. The problem with this approach is that it ignores the important fact that timing analysis is inherently path-oriented. Although net weights are a convenient way to influence layout design, it is not sufficient to produce satisfactory results.

Some algorithms for timing optimization which are path-oriented have been proposed [9] , [7]. An elegant algorithm for timing optimization, proposed by Jackson and Kuh [3], is discussed here in detail. This algorithm is concerned with performance optimization of layout for synchronous digital circuits. According to Jackson and Kuh, the main timing problems in synchronous digital circuits are problems of clock skew, long paths and short paths. *Clock skew* occurs when unequal delay from the circuit clock to sequential element clock terminals exist. The *long path* problem is concerned with ensuring that a signal arrives at all cells earlier than a required time. Finally, the *short path* timing problem occurs when a signal races through the combinational logic, arriving at a sequential element's input too early. The Jackson and Kuh algorithm is concerned with biasing the modules placement to handle the above problems.

The Jackson and Kuh algorithm approaches floorplanning hierarchically. At each level of the hierarchy, a linear programming process is carried out to obtain the cell placement. Cell overlaps exist at the end of this process because there is no module shape consideration at this stage. The linear programming process is followed by a partitioning step which distributes modules over the chip area. This cycle of linear programming and partitioning is carried out for consecutive levels of hierarchy until the chip has been subdivided into suitably small regions that

contain a predetermined small number of cells. When this procedure is finished, cells are positioned such that they agree with a predetermined structure, such as a row structure, and no overlaps exist. In order to describe the algorithm, the linear programming and partitioning procedures must be clearly explained.

The linear programming process aims to maximize an objective function value, while satisfying linear constraints equations [2]. In this process, both physical and timing properties of the layout are considered in both of objective function and the constraint equations.

Constraints are placed on the lumped capacitance for all interconnections. The capacitance per unit length for horizontal and vertical interconnections are denoted by γ and ν , respectively. For each net the capacitance of net is estimated as follows:

$$C_i = \gamma (r_i - l_i) + \nu (t_i - b_i)$$

where r_i, l_i, t_i and b_i are right, left, top and bottom dimensions of the net i 's bounding box, respectively. If total number of nets in the circuit is n , the n equations are necessary to represent the interconnection capacitance relations.

Another set of physical constraints are introduced to center the cells within the region in which they are to be placed. This is a first order linear slot constraint that forces the center of mass of all cells in a region to be identified to the center of the region. An example of this type of equation at the first level of hierarchy is:

$$\bar{x} = \frac{\sum_{i=1}^c m_i x_i}{\sum_{i=1}^c m_i}$$

where c is the total number of cells, x_i is the x coordinate of cell i in the region, m_i is the width of cell, and $\bar{x}$ is the x coordinate of the center of the region.

The timing constraints are defined as a set of delay equations. Figure 2-6 shows two cells and an interconnection net between them. For this configuration a delay equation as follows can be defined.

$$a_2 \geq a_1 + d_1 + R_1 C_1$$

In this equation, a_i is the latest signal arrival time at the input of cell i , d_i is the delay of the signal through cell i , R_i is the output resistance of cell i , and C_i is the capacitance of net i . It is assumed that cells 1 and 2 are combinational cells. If p denotes the number of terminals and c the number of cells in the circuit, and each cell has a single output, then the number of these delay equations is $p - c$. It should be mentioned that there must be special treatment for sequential elements in these delay equations. The clock skew and long path problems can be solved by satisfying these delay equations.

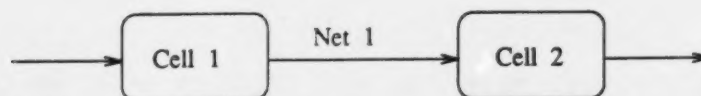


Figure 2-6: Two combinational cells to illustrate the delay equations

The objective function of the linear programming is defined as follows:

$$\psi = M - \alpha \times W$$

In this function we have:

$$\alpha = \frac{k \times \bar{R}}{n} \quad W = \sum_{i=1}^n C_i$$

Where k is a user specified weight, $\bar{R}$ is the average cell output resistance, n is the number of nets, and W is the sum of all net capacitances. The other parameter, M , represents the minimum *slack*² for all cells at the path endpoints.

² The chip clock frequency, clocking methodology and clock skew in the circuit determine the required arrival time for a cell at a path endpoint. Alternatively, the cell has an actual ar-

Maximizing M in the objective function will take care of critical paths delay, and maximizing $-\alpha \times W$ minimizes the sum of weighted net bounding box half perimeters for all nets, which tends to improve the layout's area and total wire length.

After the linear program is solved at a given level of the layout hierarchy, a partitioning stage is carried out to partition the layout region into two equal sized half-regions. Using the predefined final cell structure as a guide, an area relationship will exist between the two half-regions. In this process, cells are sorted by either the x or y position by scanning through the list. A cut line is placed at the position where the ratio of the cell area in the lower portion to the upper portion of the list equals the area ratio of the two half regions. The cut line is then dragged to the half region border, repositioning all touched cells in the adjacent half region.

The complexity of this algorithm is a function of the linear programming algorithm used. By considering all of the capacitance relations and timing equations, the number of equations and variables is linear with the size of the chip. Efficient methods for solving the system of equations can be used to yield satisfactory run time results.

2.5. Conclusions and Discussion

The Stockmeyer algorithm is an efficient algorithm and is guaranteed to generate a chip with minimum area for a given slicing structure. However, this algorithm does not address wiring optimization. The second algorithm, which was a force directed algorithm, is concerned with wiring optimization, but not directly with area. It tries to arrange the modules on the chip surface such that the modules which are connected with more nets are placed closer together. Because

rival time determined by its latest arriving signal. The long path slack at an endpoint cell is the difference between required arrival time and actual arrival time.

this algorithm is not directly concerned with area utilization, the result might not be satisfactory and it may leave too much free space. The Wong and Liu algorithm uses a cost function which is a weighted sum of chip area and wire length estimation. This algorithm is elegant and simple and more likely to achieve satisfactory results. But because of the nature of the algorithm, the result is a trade off between minimum area and minimum wire length. Another problem with this algorithm is that the wire length estimation is rather poor. The Jackson and Kuh algorithm differs from the other algorithms in that it addresses chip timing performance. As predicted, this algorithm is effective at optimizing chip performance, but the result may not necessarily be good in terms of chip area and overall wiring complexity.

In most of the floorplanning algorithms, chip area and total wire length are combined into a single cost function, as in the Wong and Liu algorithm. Because of this combination, we can expect to have room for improving the total wire length of the resulting floorplans. In fact, our problem is to modify the slicing floorplan resulting from another algorithm in order to achieve the minimum total wire length, while the chip's area or shape remains unchanged. The precise definition of this problem and two algorithms for solving the problem are discussed in the next chapter.

An important point which should be restated is that we can only estimate the total wire length of a circuit. Therefore, it is predictable that minimizing an estimate may not lead to the optimum wire length after final routing, and it does not also directly optimize the wiring area or circuit performance. However, minimizing a wire length estimate should tend to improve the overall quality of the wiring.

Chapter Three

New Algorithms For Wiring Optimization

3.1. Problem Definition

In the previous chapter, some floorplanning algorithms that attempt to minimize total wire length were reviewed. In this chapter, two closely related new approaches for minimizing total wire length in slicing floorplans are described. The objective of these algorithms is to modify an existing slicing floorplan so as to minimize the total wire length, without changing the chip area and shape. Before examining these algorithms in detail, the problem being addressed will be defined precisely.

We assume that a slicing floorplan which is described by a slicing tree is given. Let us denote the slice direction assigned to each internal node v with $C(v)$, which can take either of two values V or H representing vertical or horizontal slice directions, respectively. For each leaf node α in the tree, the assigned module will be denoted by $M(\alpha)$. The value of $C(v)$ for all internal nodes and $M(\alpha)$ for all leaf nodes are unchangeable and therefore are fixed input information to the problem.

The important point here is that, for each node in the slicing tree, the relative position of child nodes does not affect the size or shape of the floorplan. Furthermore, at the leaf levels the orientation of modules can be changed without any effect on the size and the shape of floorplan.

Let us denote the relative position of each internal node v with $A(v)$, which can take either of two values F or R . The value F indicates that the child nodes

are in the *forward* position, which is defined as the left child being to the left of the right child node for a horizontal slice, and below the right child node for a vertical slice. In contrast, the value *R* indicates that the child nodes are in *reverse* position compared to what is defined as forward position. For each leaf node α , we also denote the leaf cell orientation as $A(\alpha)$. In this case, $A(\alpha)$ can take four values *FR*, *RR*, *FU* and *RU*. The first letter in these labels represents mirroring about the vertical axis. The letter *F* represents *forward* or normal orientation, and *R* represents *reverse* or mirrored orientation. The second letter represents mirroring about the horizontal axis. The letter *R* represents *right side up* or normal orientation, and *U* represents *up side down* or mirrored orientation. Initially, $A(v)$ for each internal or leaf node v in the tree is unknown and needs to be determined by the algorithm.

The problem being addressed in this thesis can be formulated as follows:

Given:

- The circuit netlist which specifies the module terminals to be interconnected.
- A slicing floorplan with its slicing tree representation. In the slicing tree $C(v)$, for all internal nodes v , and $M(\alpha)$, for all leaf nodes α , are fixed input information.
- Modules dimensions and terminal locations.

Determine:

$A(v)$ for all internal and leaf nodes v in the tree so that the total net length of the circuit is minimum.

3.2. An Overview of Problem Solutions

Any algorithm which attempts to solve this problem must visit each node in the slicing tree, and based on the estimated total wire length, must make a

decision about $A(v)$. The total wire length estimate which we use is the sum of all net bounding box half perimeters. This estimate is easy and fast to compute, and is accurate enough for our purposes. The value of $A(v)$ for each node in the slicing tree describes a configuration of the floorplan which is referred to as a problem state. In general, the state space of a problem is represented by a directed graph in which each node represents a problem state, and each directed edge represents a step in the problem solving process. Because of the nature of our problem, this graph is a tree, and the root of the tree represents the initial state of the problem where $A(v)$ is unknown for all of the nodes in the slicing tree. At every other node in the state space tree, $A(v)$ for some or all of the nodes in the slicing tree is known. The depth of the state space tree is equal to the number of nodes in the slicing tree. The algorithm finds a solution path through this tree from the root, which is the initial state, to one of leaf nodes which corresponds to a floorplan configuration with the minimum total wire length. Figure 3-1 illustrates a slicing tree and its state space tree representation. The floorplan in this figure is a very simple one and consists of only two modules which are placed next to each other. The slicing tree for this floorplan has only three nodes.

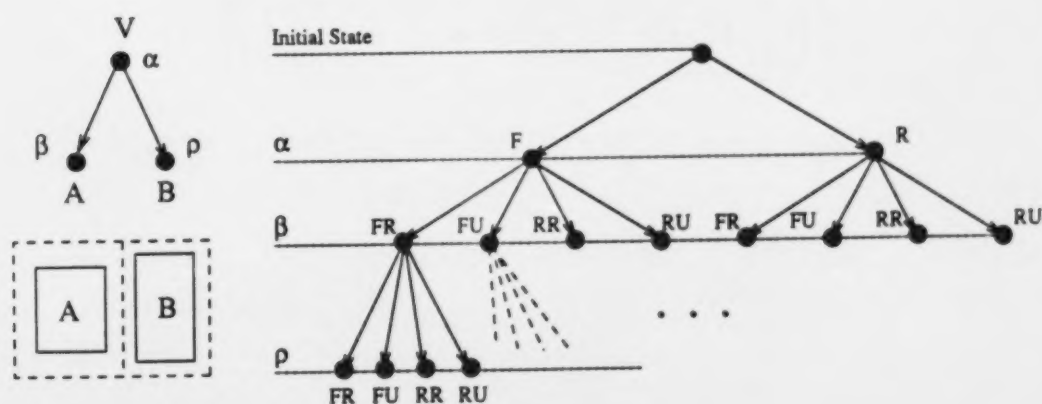


Figure 3-1: The problem state space tree

One method of solving this problem is to traverse all possible solution paths in the state space tree, and then select the one with the lowest total wire length as the final solution. Although this method, which is called *exhaustive search*, finds the best possible solution, it cannot be performed within a practical length of time for problem instances of interesting size. In fact, for a floorplan with n modules, the state space tree has 2^{3n-1} different paths to examine; thus the state space tree grows exponentially with the number of modules.

An interesting technique that reduces the search complexity is called *Branch and Bound*. Branch and Bound traverses paths one at a time, keeping track of the best path found so far. The total wire length associated with this path is used as a bound on future paths to be searched. As the search proceeds along a path, the algorithm at each state places a lower bound on the total wire length which can be achieved in the solution rooted at that state. If the lower bound is greater than the best wire length found so far, the algorithm eliminates the next state and all of its possible extensions from the state space tree, and hence it greatly reduces the search complexity. It should be mentioned that although the branch and bound reduces search complexity considerably, it may still examine a large number of paths.

The important calculation in the branch and bound is how to find a lower bound on the total wire length for a state of the problem in which the value of $A(v)$ for some of the nodes in the slicing tree is not determined. Based on the relative position assigned to these undecided nodes, different configurations with different total wire lengths may be obtained. The matter of finding the lower bound will be discussed later in this chapter. The value of the lower bound is updated, along a path in the state space tree, once the relative position of nodes in the slicing tree is determined.

Another strategy for reducing search complexity is to find the solution path according to a heuristic or rule. One simple rule that may be used is to select, at each state from all possible states at the next level, the state with the lowest value of lower bound on the total wire length. This algorithm is highly efficient, since there is only one path to be tried. Clearly, this method does not necessarily leads to the optimum solution, because the decisions are based on local information.

In the rest of this chapter, we will discuss the details of a branch and bound and a heuristic algorithm for solving our problem. The motivation for implementing the branch and bound algorithm for our problem is that it might be efficient for floorplans of practical size. Alternatively, the heuristic algorithm might be able to come up with reasonably good solutions very quickly.

3.3. A New Branch and Bound Algorithm For Wiring Optimization

In this section the details of a branch and bound algorithm for our problem are discussed. As was pointed out before, the key to a successful branch and bound algorithm is an efficient and accurate way of computing a lower bound on the total wire length. In order to calculate the required lower bound, denoted by P_{low} , and update it along different paths in the state space tree, the slicing tree must be processed prior to running branch and bound algorithm. Therefore, before discussing the structure of the algorithm, we will consider the information which is required at each node in the slicing tree and then describe how this information is provided by a pre-processing algorithm.

3.3.1. The Pre-Processing Algorithm

The pre-processing algorithm traverses the slicing tree, in bottom-up fashion, to calculate an initial value of P_{low} . The following information is provided for

each node v (internal or leaf node) in the slicing tree:

- $xmin(n_j, v)$, $ymin(n_j, v)$
- $wmin(n_j, v)$, $hmin(n_j, v)$
- $terminals(n_j, v)$
- $List(v)$
- $width(v)$, $height(v)$

The definition of this information and how they are calculated are discussed in the following paragraphs.

$xmin(n_j, v)$ & $ymin(n_j, v)$:

$xmin(n_j, v)$ is defined as the minimum of either of x_1 or x_2 , where we have:

x_1 = the minimum possible distance from the left side of v 's rectangle to the furthest of n_j 's terminals in v .

x_2 = the minimum possible distance from the right side of v 's rectangle to the furthest of n_j 's terminals in v .

Figure 3-2 illustrates an example of net n_j 's bounding box which encloses n_j 's terminals in rectangle v (rectangle corresponds to node v . The value of x_1 and x_2 are specified in this figure. In this example, since x_1 is smaller than x_2 , we have: $xmin(n_j, v) = x_1$

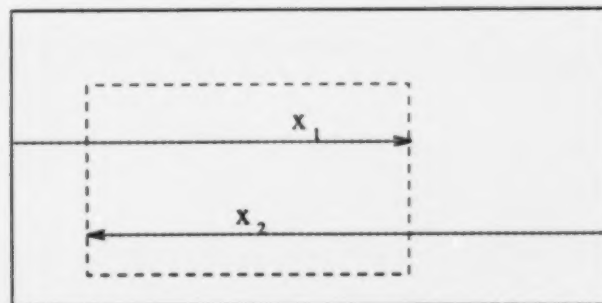


Figure 3-2: Definition of $xmin()$

Similarly, $ymin(n_j, v)$ is defined as the minimum of either of y_1 or y_2 , where we have:

y_1 =the minimum possible distance from the bottom side of v 's rectangle to the furthest of n_j 's terminals in v .

y_2 =the minimum possible distance from the top side of v 's rectangle to the furthest of n_j 's terminals in v .

For a leaf node v , these values can be calculated from the leaf cell geometry. For an arbitrary net n_j , we can find the location of the rightmost and leftmost terminals within a leaf rectangle, then $xmin(n_j, v)$ is the minimum of the distances from the rightmost terminal to the rectangle's left side, or the distance from the leftmost terminal to the rectangle's right side. The same technique can also be used for calculating $ymin(n_j, v)$.

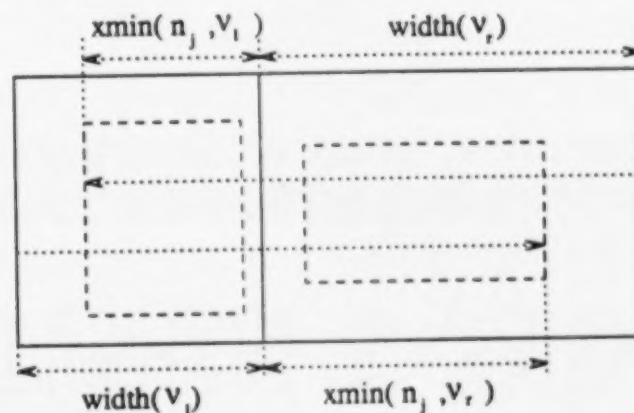


Figure 3-3: Calculation of $xmin()$ for an internal node

For an internal node v_p , let us assume net n_j has terminals in both of v_p 's children v_l and v_r . Calculation of $xmin(n_j, v_p)$, for the case where v_p corresponds to a vertical slice, is illustrated in Figure 3-3. By considering the situation in this figure, it can be seen that the minimum distance from the right side of v_p to the furthest terminal of net n_j is $xmin(n_j, v_l) + width(v_r)$.

Similarly, the minimum distance from the left side of v_p to the furthest terminal of net n_j is $xmin(n_j, v_r) + width(v_l)$. Note that $width()$ describes the width of the rectangle corresponding to a node. Therefore, $xmin(n_j, v_p)$ can be calculated from the following equation:

$$xmin(n_j, v_p) = \min(width(v_l) + xmin(n_j, v_r), width(v_r) + xmin(v_l, n_j))$$

On the other hand, the vertical slice has no effect on the vertical extent of the net bounding box. Thus we have:

$$ymin(n_j, v_p) = \max(ymin(n_j, v_l), ymin(n_j, v_r))$$

If the slice direction corresponding to v_p is horizontal, the only difference in the above equations is the change in the role of $xmin()$ and $ymin()$, and the replacement of $width()$ with $height()$, where $height()$ describes the height of the rectangle corresponding to a node.

The value of $xmin(n_j, v_p)$ and $ymin(n_j, v_p)$, when n_j has terminals in only one of v_p 's children, is equal to $xmin()$ and $ymin()$ in that child node. For instance, if net n_j has terminals only in v_l , then $xmin(n_j, v_p)$ is equal to $xmin(n_j, v_l)$, and $ymin(n_j, v_p)$ is equal to $ymin(n_j, v_l)$.

wmin(n_j, v) & hmin(n_j, v):

$wmin(n_j, v)$ is defined as the minimum width of net n_j 's partial bounding box¹ enclosing net n_j 's terminals within rectangle v . The size of partial bounding boxes for leaf nodes does not change with different module orientations. But for internal nodes, the relative position of child nodes and their descendants affects the size of net bounding boxes, and for different module arrangements, a net's partial bounding box may have different sizes. $wmin(n_j, v)$ specifies the minimum

¹ The net partial bounding box is defined as the bounding box which encloses some but not all of the nets' terminals.

possible width of net n_j 's partial bounding box that is achievable by some arrangement of modules inside v 's rectangle. $hmin(n_j, v)$ is similar to $wmin(v, n_j)$ and is defined as the minimum possible height of net n_j 's partial bounding box.

Calculation of $wmin()$ and $hmin()$ is similar to that of $xmin()$ and $ymin()$. For a leaf node v , the distance between the rightmost and leftmost terminals of net n_j inside v is $wmin(n_j, v)$. Similarly, $hmin(n_j, v)$ is the distance between topmost and bottommost terminals of n_j lying in v .

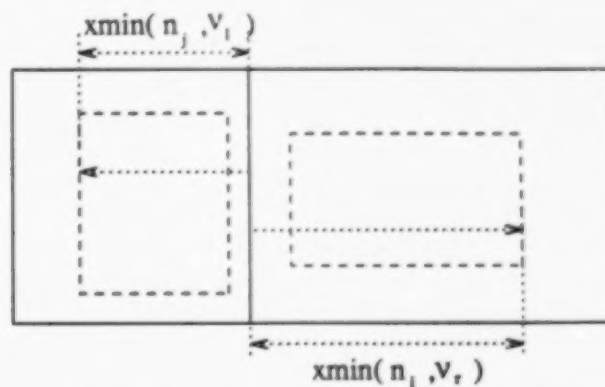


Figure 3-4: Calculation of $wmin()$ for an internal node

To calculate the value of $wmin()$ and $hmin()$ for an internal node, let us consider the situation in Figure 3-4, where v_p corresponds to a vertical slice and net n_j is a common net between v_p 's children v_l and v_r . According to the definitions, the minimum possible extent of net n_j to the right of the slice line will be $xmin(n_j, v_r)$, and similarly the minimum possible extent of net n_j to the left of the slice line is $xmin(n_j, v_l)$. Thus we have:

$$wmin(n_j, v_p) = xmin(n_j, v_l) + xmin(n_j, v_r)$$

Because the vertical slice has no effect on the horizontal extent of net n_j , we have:

$$hmin(n_j, v_p) = \max(hmin(n_j, v_l), hmin(n_j, v_r))$$

A similar method can be used for calculation of $wmin(n_j, v_p)$ and $hmin(n_j, v_p)$, when the slice direction at v_p is horizontal.

Analogous with $xmin()$ and $ymin()$, the value of $wmin(n_j, v_p)$ or $hmin(n_j, v_p)$, where net n_j has terminals in only one of v_p 's children, say v_l , is equal to $wmin(n_j, v_l)$ and $hmin(n_j, v_l)$, respectively.

terminals(n_j, v) :

This parameter is defined as the number of terminals of net n_j lying inside rectangle v . For the leaf nodes, this parameter can be determined from the circuit netlist. For internal nodes it can be obtained by adding the number of nets' terminals in the child nodes.

List(v) :

List(v) is a netlist associated with node v . If v is an internal node, then the list contains all the nets which have some, but not all, of their terminals lying in the composite rectangle corresponding to node v . If v is a leaf node, the list contains all the nets which are attached to the module assigned to v . For both leaf nodes and internal nodes, a net n_j is omitted from *List(v)* if all of n_j 's terminals lie within rectangle v . To improve the efficiency of the branch and bound algorithm, this list is sorted in nondecreasing order of net id-numbers; also, all of the above listed parameters are only calculated for the nets in this list.

The procedure for constructing *List()* for a leaf node is different from that for an internal node. For a leaf node v , the nets in *List(v)* can be easily extracted from the circuit netlist. To explain how a list is constructed for an internal node, we consider again a node v_p with two children v_l and v_r , whose netlists *List(v_l)* and *List(v_r)* are already constructed. Since *List(v_l)* and *List(v_r)* are sorted, these lists are merged and the nets which satisfies the requirements will be added to *List(v_p)*. The following pseudo-code describes the construction of *List(v_p)* in

details. As the pseudo-code demonstrates, as soon as a net is added to $List(v_p)$, its attributes $terminals(n_j, v_p)$, $xmin(n_i, v_p)$, $ymin(n_i, v_p)$, $wmin(n_i, v_p)$ and $hmin(n_i, v_p)$ are calculated and saved.

```

Construct (  $v_p$  )
 $v_p$  : an internal node in the slicing tree.
{
 $v_l = v_p$ 's left child.
 $v_r = v_p$ 's right child.

For all common nets  $n_i$  in  $List(v_l)$  and  $List(v_r)$ 
{
    if( $terminals(n_i, v_l) + terminals(n_i, v_r) <$ 
    total number of  $n_i$ 's terminals)
    {
        Add  $n_i$  to  $List(v_p)$ 

         $terminals(n_i, v_p) = terminals(n_i, v_l) + terminals(n_i, v_r)$ 

        Calculate  $xmin(n_i, v_p), ymin(n_i, v_p), hmin(n_i, v_p)$ 
        and  $wmin(n_i, v_p)$  and save them.
    }
    else
        Calculate minimum dimensions of net  $n_i$ 's
        bounding box and save them.
}

For all nets  $n_j$  in  $List(v_l)$  which are not common with  $List(v_r)$ 
{
    if( $terminals(n_j, v_l) <$  total number of  $n_j$ 's terminals)
    {
        Add  $n_j$  to  $List(v_p)$ 

         $terminals(n_j, v_p) = terminals(n_j, v_l)$ 

        Calculate  $xmin(n_j, v_p), ymin(n_j, v_p), hmin(n_j, v_p)$ 
        and  $wmin(n_j, v_p)$  and save them.
    }
}

For all nets  $n_j$  in  $List(v_r)$  which are not common with  $List(v_l)$ 
{
    Perform calculations similar to that above for  $n_j$ 
}
}

```

In the above procedure, one issue should be discussed. When the procedure detects a net n_j which is common between v_l and v_r , but which does not satisfy the number of terminals criterion, the net will not be added to the $List(v_p)$. But

in this case, net n_j lies entirely inside v_p , and at this point, we can find the minimum width and height of n_j 's overall bounding box which encloses all of n_j 's terminals in the entire floorplan. Calculation of the minimum width and height of the net n_j 's bounding box is exactly like the calculation of $hmin()$ or $wmin()$,⁴ For the case where v_p corresponds to a vertical slice is as follows:

$$\begin{aligned} net\_width(n_j) &= xmin(n_j, v_l) + xmin(n_j, v_r) \\ net\_height(n_j) &= \max(hmin(n_j, v_l), hmin(n_j, v_r)) \end{aligned}$$

where $net\_width(n_j)$ and $net\_height(n_j)$ are the minimum width and height of n_j 's bounding box. After the entire slicing tree has been pre-processed, these values for all the nets in the circuit are added to obtain the initial value of P_{low} .

width(v), height(v) :

$width(v)$ and $height(v)$ are the width and height of the floorplan rectangle corresponding to the node v . In contrast to the listed parameters above, $width()$ and $height()$ are calculated in a different manner. These values are calculated by processing the slicing tree in two passes. On the first pass, the slicing tree is processed in bottom-up fashion, and the minimum possible width and height for each node are obtained. To explain how first pass values of $width()$ and $height()$ are calculated for an internal node, consider a node v_p with vertical slice direction. Because the algorithm moves bottom-up in the slicing tree, $width()$ and $height()$ are known at v_r and v_l , which are v_p 's right and left children respectively. $width(v_p)$ and $height(v_p)$ are calculated as follows:

$$\begin{aligned} width(v_p) &= width(v_l) + width(v_r) \\ height(v_p) &= \max(height(v_l), height(v_r)) \end{aligned}$$

⁴ In fact, the minimum dimensions of net n_j 's bounding box is $hmin(n_j, v_p)$ and $wmin(n_j, v_p)$.

The height and width of nodes with a horizontal slice direction are calculated similarly.

Following the first pass, the slicing tree is processed from top to bottom in order to obtain the final values of *height()* and *width()*. The first pass values of *height()* and *width()* for the root are not changed. For any internal node, the first pass values of *height()* and *width()* for its children are available. If the internal node v_p has a vertical slice direction, and the height of v_p 's children are unequal, then the smaller height will be updated to the larger height. On the other hand, if the sum of the width of v_p 's children is not equal to v_p 's width, the width of both v_p 's children will be updated such that they still have the same ratio, and the sum of them is equal to the v_p 's width. Similar calculations are carried out for the internal nodes whose slice directions are horizontal.

The following pseudo-code summarizes the pre-processing algorithm which has been discussed in this section:

```

Pre-Process (  $v_p$  )
 $v_p$  : a node in the slicing tree.
{
    If (node  $v_p$  is an internal node)
    {
         $v_l$  = node  $v_p$  left child.
         $v_r$  = node  $v_p$  right child.

        Pre-Process (  $v_l$  )
        Pre-Process (  $v_r$  )

        Construct( $v_p$ )
    }
    else /*  $v_p$  is a leaf node. */
    {
        List( $v_p$ ) = all the nets attached to  $M(v_p)$ 

        For each net  $n_j$  in List( $v_p$ )
        Calculate terminals ( $n_j, v_p$ ),  $xmin(n_j, v_p)$ ,  $ymin(n_j, v_p)$ 
         $wmin(n_j, v_p)$  and  $hmin(n_j, v_p)$ 
    }
}

```

]

This procedure calls itself recursively and with this technique it goes to the leaf levels of slicing tree, and then returns to the top. In order to process the whole slicing tree, the procedure must be called for the root of the tree.

3.3.2. The Branch and Bound Algorithm

After the whole slicing tree has been pre-processed, the branch and bound algorithm can be applied to the problem. A pseudo-code description of the branch and bound algorithm, BandB(), is as follows:

```

BandB(queue)
queue: contains all the nodes in the slicing tree in
parent first order.
{
  If (queue is not empty)
  {
     $v_p$  = the node at the head of the queue.
    If ( $v_p$  is an internal node)
    {
       $A(v_p) = F$ 
      Slice Cal( $v_p$ )
      Update Internal( $v_p$ )
      If ( $P_{low} < P_{best}$ )
      {
        Remove  $v_p$  from the queue.
        BandB(queue)
        Undo all of the changes to the queue.
      }
      Undo all of the changes to  $P_{low}$ 
      and the minimum bounding boxes.
       $A(v_p) = R$ 
      Repeat procedure similar to that above.
    }
    else /* If  $v_p$  is a leaf node */
    {
       $A(v_p) = FR$ 
      Update Leaf( $v_p$ )
      If ( $P_{low} < P_{best}$ )
      {
        Remove  $v_p$  from the queue.
      }
    }
  }
}

```

```

        BandB(queue)
        Undo all of the changes to the queue.
    }
    Undo all of the changes to  $P_{low}$ 
    and the minimum bounding boxes.

     $A(v_p) = RR$ 
    Repeat procedure similar to that above.
     $A(v_p) = FU$ 
    Repeat procedure similar to that above.
     $A(v_p) = RU$ 
    Repeat procedure similar to that above.
}

}
else /* queue is empty */
{
    If( $P_{low} < P_{best}$ )
    {
         $P_{best} = P_{low}$ 
        Save the current relative position for all the nodes
        as the best solution found so far.
    }
}
}

```

The above procedure operates by recursively processing a queue of slicing tree nodes. The nodes can appear in the queue in different orders, provided that each node precedes either of its children. Two specific orders in which we are interested are depth-first and breadth-first. In the depth first order, when a node is examined, all of its children and its descendants are examined before any of its siblings. For breadth first, nodes are visited in a level by level fashion. Although the final configuration resulting from the algorithm for both of these orders must be the same, the ability to cut the branches in the search tree may be different. This issue is addressed in the next chapter.

Two parameters are important in the branch and bound algorithm. The first parameter is the lower bound on the total wire length, which is denoted by P_{low} . The initial value of P_{low} is produced by the pre-processing algorithm (see page

44). The second parameter is the smallest total wire length encountered by the algorithm up to the current state in the search. This parameter, which is denoted by P_{best} , is used to cut the branches in the state space tree and speed up the search. The initial value of P_{best} can be obtained by assigning arbitrary relative positions to all nodes in the slicing tree and then calculating the resulting total wire length. These arbitrary relative positions can also be the result of a sub-optimal search for a solution.

The branch and bound algorithm starts from the problem initial state, where the queue of slicing tree nodes is full. Let us assume that node v_p is at the head of the queue. First, $BandB()$ assigns the relative position F to v_p and updates the value of P_{low} based on this decision. In $BandB()$, $Update\_Internal()$ and $Slice\_Cal()$ are the functions called for updating P_{low} . The details of these functions will be explained later. If P_{low} is less than P_{best} , the algorithm may be on the right track to the optimum solution. Therefore, $BandB()$ removes v_p from the queue and recursively calls itself. Alternatively, if P_{low} is greater than P_{best} , then we cannot obtain the optimum solution with this relative position and $BandB()$ does not call itself. In either of these cases, the changes made to P_{low} and the queue are undone. Then the relative position R is assigned to v_p and the above procedure is repeated.

If the node at the head of the queue is a leaf node, a procedure similar to that above is used, with the only difference being that $BandB()$ tries all four possible relative positions for a leaf node. In this case, $Update\_Leaf()$ is the function called for updating P_{low} , and this function will be explained in detail later.

When the queue becomes empty, P_{low} is the actual total half perimeter of the circuit. In this situation, if P_{low} is less than P_{best} , then the current solution is better than the current best solution. In this case, $BandB()$ sets P_{best} equal to P_{low}

and saves the current relative position for all the nodes in the slicing tree as the best solution found so far.

An important issue in the branch and bound algorithm is computing P_{low} , the lower bound on the total wire length of the circuit. As was explained earlier, the pre-processing algorithm obtains the value of P_{low} by summing the minimum width and height of all the net bounding boxes in the circuit. In other words, the lower bound on the total wire length of the circuit is the sum of the lower bounds on half perimeter of the individual nets in the circuit. To update P_{low} , BandB() keeps track of the lower bound on each net bounding box half perimeter individually. For each net n_j , a *minimum bounding box* is maintained which is defined as the smallest possible bounding box that can enclose net n_j 's terminals, given that relative positions have been assigned to some of the nodes in the slicing tree. Net n_j 's minimum bounding box is described by four parameters $r(n_j)$, $l(n_j)$, $t(n_j)$ and $b(n_j)$; these four parameters give the right, left, top and bottom boundary locations, respectively, of net n_j 's minimum bounding box. For instance $r()$ and $l()$ are defined as follows:

$r(n_j)$ = *Smallest possible value of the right boundary of net n_j 's bounding box*

$l(n_j)$ = *Largest possible value of the left boundary of net n_j 's bounding box*

In other words, the right boundary of n_j 's bounding box is pushed to the leftmost possible position, and this position is denoted by $r(n_j)$. Similarly, the left boundary is pushed to the rightmost possible position and this position is denoted by $l(n_j)$. The interesting feature of the minimum bounding boxes is that, in defining $r()$ and $l()$, we don't care about the relative position of right and left boundaries. In other words, in some situations we may have $r() < l()$.

As the relative position decisions are made in the slicing tree, the minimum bounding box dimensions are updated, and if the update has an impact on P_{low} ,

then P_{low} is also updated. Eventually, when the queue of slicing tree nodes becomes empty, the relative position for all the nodes in the slicing tree is determined. In this situation, the minimum bounding boxes become the actual bounding boxes for the nets, and P_{low} becomes the actual total half perimeter of the floorplan.

When a decision about node's relative position is made, the minimum bounding boxes of the nets which are affected must be updated. For a leaf node, all of the nets which are attached to the corresponding leaf cell must be considered. These nets are available from the $List()$ structure for that leaf node. For an internal node v_p , nets whose terminals lie entirely in one of v_p 's children are special cases and need not to be considered by the updating procedure. The relative position at v_p can affect the location of these nets' bounding boxes, but it has no effect on their size. The nets which must be considered by the updating procedure are those in $List(v_l)$ and $List(v_r)$ where v_l and v_r are v_p 's left and right children.

The procedure for updating minimum bounding boxes is different for leaf nodes and internal nodes. First, we will discuss the updating procedure for internal nodes. Let us assume v_p is an internal node where $C(v_p)=V$ and the algorithm has assigned $A(v_p)=F$. According to the definition, for an arbitrary net n_j in $List(v_l)$, the closest possible distance between net n_j 's partial bounding box right boundary and the left side of v_l is $xmin(n_j, v_l)$. Since the position of the slice line and the value of $xmin()$ are known, the leftmost possible position of n_j 's partial bounding box right boundary is also known. This position is illustrated in Figure 3-5 (a). If this value is larger than the current value of $r(n_j)$, $r(n_j)$ is updated to the new value. Similarly, the closest possible distance from net n_j 's partial bounding box left boundary to the right side of v_l is also $xmin(n_j, v_l)$. Therefore, the rightmost possible position of n_j 's minimum bounding box left

boundary is known. This position is illustrated in Figure 3-5 (b). If this value is smaller than the current value of $l(n_j)$, $l(n_j)$ is updated.

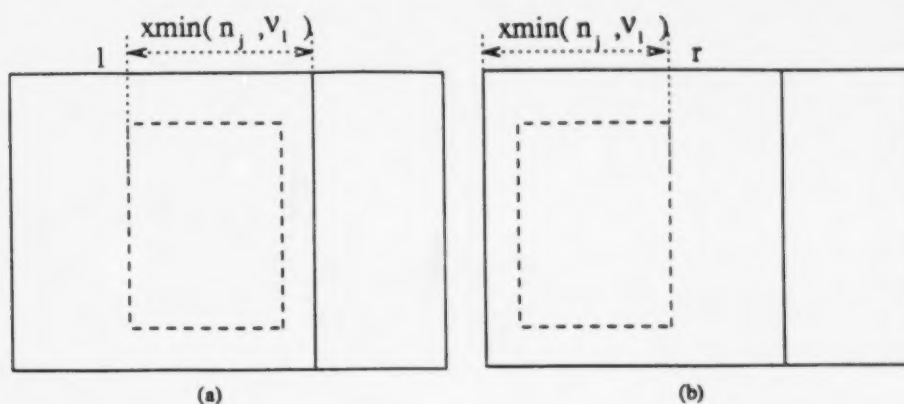


Figure 3-5: Updating $r(n_j)$ and $l(n_j)$

Summarizing, the values of $r(n_j)$ and $l(n_j)$ are updated according to the following equations:

$$\begin{aligned} \text{new\_}r(n_j) &= \max(\text{old\_}r(n_j), \text{slice}(v_p) - \text{width}(v_l) + \text{xmin}(n_j, v_l)) \\ \text{new\_}l(n_j) &= \min(\text{old\_}l(n_j), \text{slice}(v_p) - \text{xmin}(n_j, v_l)) \end{aligned}$$

In the above equations $\text{xmin}(n_j, v_l)$ and $\text{width}(v_l)$ are already determined by the pre-processing algorithm. $\text{slice}(v_p)$ is the x or y position of the slice line, depending on the slice direction. The origin of these coordinates is the left-bottom corner of the whole floorplan, and therefore $\text{slice}(v_p)$ describes the absolute position of slice line in the floorplan. Similarly, for each net n_i in $\text{List}(v_r)$, $r(n_i)$ and $l(n_i)$ are updated according to the following equations:

$$\begin{aligned} \text{new\_}r(n_i) &= \max(\text{old\_}r(n_i), \text{slice}(v_p) + \text{xmin}(n_i, v_r)) \\ \text{new\_}l(n_i) &= \min(\text{old\_}l(n_i), \text{slice}(v_p) + \text{width}(v_r) - \text{xmin}(n_i, v_r)) \end{aligned}$$

The relative position at v_p has no effect on the y position of the minimum

bounding boxes when the $C(v_p)=V$. Therefore, only $r()$ and $l()$ boundaries of bounding boxes need to be updated and $t()$ and $b()$ remain unchanged. Alternatively, if $C(v_p)=H$, then only the vertical extent of the minimum bounding boxes are affected. In this case, only $t()$ and $b()$ are updated in a manner similar to $r()$ and $l()$ for each net in $List(v_l)$ and $List(v_r)$, using $ymin()$, $height()$ and $slice()$ information.

A question which must be answered is "What is the initial value of the minimum bounding box boundaries?". To answer this question, let us assume that net n_j lies entirely in some node v_l whose parent is v_p . Obviously, a relative position assigned to v_p has a great effect on the position of n_j 's minimum bounding box. If v_l is the left child of v_p , then assigning forward relative position to v_p places the net n_j 's minimum bounding box in the left of the slice line, and assigning the reverse relative position at v_p places the bounding box in to the right of the slice line. Although the relative position assignment at v_p changes the location of the minimum bounding box, it provides no information about the size of the minimum bounding box. For all v_p 's ancestors, net n_j also lies entirely in one of their children and the relative position of these nodes also will not change the size of n_j 's minimum bounding box.

Initially, the relative position of all the nodes in the slicing tree is undetermined, and therefore the initial values of $r()$, $l()$, $t()$ and $b()$ for all nets are unknown. The problem here is that if these values are initially unknown, how we can update them based on the technique explained earlier. To solve this problem, the algorithm initially assigns the NULL value to $r()$, $l()$, $t()$ and $b()$ for all of the nets in the circuit. The NULL value indicates that the position of the minimum bounding box is unknown. Every time that the algorithm makes a decision about the relative position of a node v_p , it looks to its children net lists. Each net which is

considered for the first time has NULL boundaries, and its boundaries at that point become defined. To clarify this point, let us assume that node v_p is an internal node with $C(v_p)=V$ and $A(v_p)=F$, and n_j is a net in $List(v_l)$, where v_l is v_p 's left child. Similar to the procedure for updating the minimum bounding boxes, $r(n_j)$ and $l(n_j)$ become defined as follows:

$$\begin{aligned} r(n_j) &= slice(v_p) - width(v_l) + xmin(n_j, v_l) \\ l(n_j) &= slice(v_p) - xmin(n_j, v_l) \end{aligned}$$

By examining the above equations, it can be easily verified that, for $xmin(n_j, v_l) < width(v_l)/2$, the value of $r(n_j)$ is less than $l(n_j)$. In this case, we do not care that the minimum bounding box has negative dimensions.

Although the method of updating the minimum bounding boxes has been explained, the matter of updating P_{low} has not been clearly addressed. Let us come back to the definition of P_{low} . P_{low} is the sum of the minimum bounding box half perimeters for all the nets in the circuit. As far as each individual net is concerned, there is an arrangement of modules which leads to the minimum width and height for that net's bounding box and these minimum values were obtained by the pre-processing algorithm. Therefore, the initial value of P_{low} obtained by the pre-processing may be an unrealistic value for the total half perimeter of the circuit. To update P_{low} , the amount of change which is made to the minimum bounding box dimensions when they are updated must be added to P_{low} . However, when $r(n_j) < l(n_j)$ or $t(n_j) < b(n_j)$, no changes are made to P_{low} . As well, P_{low} is only updated when the resulting dimensions become larger than the pre-processing minimum dimensions.

A pseudo-code description for the whole procedure of updating minimum bounding boxes and P_{low} for an internal node appears as follows:

```

Update Internal(  $v_p$  )
 $v_p$  : an internal node in the slicing tree with  $C(v_p)=V$ .
{
 $v_l = v_p$ 's left child.
 $v_r = v_p$ 's right child.

For each net  $n_j$  in List( $v_l$ )
If ( $r(n_j)$  and  $l(n_j) \neq \text{NULL}$ )
{
    old_r =  $r(n_j)$ 
    old_l =  $l(n_j)$ 

 $r(n_j) = \max(\text{old\_r}, \text{slice}(v_p) - \text{width}(v_l) + \text{xmin}(n_j, v_l))$ 
 $l(n_j) = \min(\text{old\_l}, \text{slice}(v_p) - \text{xmin}(n_j, v_l))$ 

    If ( $\text{old\_r} - \text{old\_l} > \text{net\_width}(n_j)$ )
         $P_{\text{low}} = P_{\text{low}} + r(n_j) - \text{old\_r} + \text{old\_l} - l(n_j)$ 
    else if ( $r(n_j) - l(n_j) > \text{net\_width}(n_j)$ )
         $P_{\text{low}} = P_{\text{low}} + r(n_j) - l(n_j) - \text{net\_width}(n_j)$ 
}
else/*  $r(n_j)$  and  $l(n_j) = \text{NULL}$  */
{
     $r(n_j) = \text{slice}(v_p) - \text{width}(v_l) + \text{xmin}(n_j, v_l)$ 
     $l(n_j) = \text{slice}(v_p) - \text{xmin}(n_j, v_l)$ 
}
Repeat procedure similar to the above for each net  $n_j$  in List( $v_r$ )
}

```

The same techniques can be used to update the minimum bounding boxes and P_{low} for a leaf node. The main difference here is that for each net in List() associated with the leaf node, the actual position of net terminals will be used to define or update the minimum bounding boxes. The actual position of terminals are calculated based on the cell orientation. This procedure is illustrated by the following pseudo-code:

```

Update Leaf(  $v_p$  )
 $v_p$  : a Leaf node in the slicing tree.
{

For each net  $n_j$  in List( $v_p$ )
maxx = absolute x position of  $n_j$ 's rightmost terminal.
minx = absolute x position of  $n_j$ 's leftmost terminal.
maxy = absolute y position of  $n_j$ 's topmost terminal.
miny = absolute y position of  $n_j$ 's bottommost terminal.
}

```



```

If ( $r(n_j)$  and  $l(n_j) \neq \text{NULL}$ )
{
    old_r =  $r(n_j)$ ;
    old_l =  $l(n_j)$ ;

     $r(n_j) = \max(\text{old\_r}, \text{maxx})$ 
     $l(n_j) = \min(\text{old\_l}, \text{minx})$ 

    If ( $\text{old\_r} - \text{old\_l} > \text{net\_width}(n_j)$ )
         $P_{low} = P_{low} + r(n_j) - \text{old\_r} + \text{old\_l} - l(n_j)$ 
    else if ( $r(n_j) - l(n_j) > \text{net\_width}(n_j)$ )
         $P_{low} = P_{low} + r(n_j) - l(n_j) - \text{net\_width}(n_j)$ 
}
else/*  $r(n_j)$  and  $l(n_j) = \text{NULL}$  */
{
     $r(n_j) = \text{maxx}$ 
     $l(n_j) = \text{minx}$ 
}

Update  $t(n_j)$ ,  $b(n_j)$  and  $P_{low}$  using maxy and miny
}

```

The final point which should be discussed in the branch and bound algorithm is the function `Slice_Cal()`. In order to update the minimum bounding boxes, and subsequently P_{low} , for an internal node, the position of the slice line is required. Whenever a relative position is assigned to an internal node v_p , this function is called to calculate the slice line position with respect to the left-bottom corner of the floorplan. This function calculates the slice line position based on v_p 's children's dimensions, which are already determined by the pre-processing algorithm, and the coordinates of v_p 's left-bottom corner. Therefore, in addition to the position of v_p 's slice line, this function calculates the coordinates of its children's left-bottom corner.

3.4. A Heuristic Algorithm for Wiring Optimization

In terms of a state space search, heuristics are interpreted as rules for choosing those branches in a state space tree that are most likely to lead to an accept-

able problem solution. A simple way to implement a heuristic search was described earlier in this chapter. In this algorithm, we try all possible next states at the next level of state space tree and evaluate the total wire length estimate for each one. Then, the state with lowest total wire length estimate is selected and the algorithm proceeds. This strategy is called a *best-first search*.

In order to implement a best-first search for our problem, we can use the lower bound on the total wire length of the circuit, the same as the one in the branch and bound algorithm. A procedure is required which evaluates P_{low} at each node of the slicing tree for different relative positions, and saves the relative position with the lowest P_{low} . This procedure can be coded using the available functions from the branch and bound algorithm as follows:

```

Best( $v_p$ )
 $v_p$ : an arbitrary node in the slicing tree
{
    If ( $v_p$  is an internal node)
    {
        A( $v_p$ ) = F
        Slice_Cal( $v_p$ )
        Update_Internal( $v_p$ )
        Save  $P_{low}$  for the forward relative position

        Undo all of the changes to the  $P_{low}$ 
        and the minimum bounding boxes.

        Repeat similar procedure to that above for
        the relative position R

        Save the relative position with the lower  $P_{low}$ 
        and update  $P_{low}$  accordingly.
    }
    else /*  $v_p$  is a leaf node */
    {
        A( $v_p$ ) = FR
        Update_Leaf( $v_p$ )
        Save  $P_{low}$  for the forward right side up orientation
    }
}

```

*Undo all of the changes to the P_{low}
and the minimum bounding boxes.*

*Repeat procedure similar to that above for the
relative positions RR, FU and RU.*

*Save the orientation with the lowest P_{low}
and update P_{low} accordingly.*

}
}

By using this procedure, the best-first search through the problem state space can be implemented as follows:

Best_First(queue)
*queue: contains all the nodes in the slicing tree in
parent first order.*
{
While (queue is not empty)
{
v_p = The node at the head of the queue.
Best(v_p)
Remove v_p from the queue.
}
}

It can be seen that *Best()* resembles the *BandB()* procedure in the previous section. In fact, the best-first search code can be rewritten using *BandB()* as follows:

Best_First(queue)
*queue: contains all the nodes in the slicing tree in
parent first order.*
{
While (queue is not empty)
{
*Remove one element from the head of the main queue
 and put it into Heu-queue.*
P_{best} = A Big Number
}
}

BandB(Heu_queue)

*Update P_{low} and the minimum bounding boxes with
the best relative position saved for the node
in the Heu_queue.*

}
}

In order to utilize the BandB() procedure, a new queue called *Heu_queue* is defined. This queue contains only one node from the slicing tree which is the current node being examined by the heuristic. Initially, *Heu_queue* is empty and one node from the main queue is removed and added to *Heu_queue*. Then P_{best} is set to a big number, so that when the BandB() is called, the "if" conditions at the beginning of BandB() always succeed and BandB() tries all possibilities for the node in the *Heu_queue*. When the BandB() procedure is finished, the best possible relative position is saved. Then the value of P_{low} and net bounding box boundaries are updated accordingly. Finally, the node in *Heu_queue* is removed and another node from the main queue is put into the *Heu_queue* and the whole procedure is repeated until the main queue becomes empty.

The main problem with the best-first search is that it can not predict the behavior of a problem along the state space search. In other words, the best relative position at each level may not to be the best in the absolute sense. One way to overcome this problem is to go along all different paths in the state space originating from the current node. This method is obviously the same as the branch and bound search which we have already implemented. A compromise between the best-first and the branch and bound search is that, instead of going down all the paths in the state space tree, we try all the paths down to a certain depth from the current node. Although this method, which is called a *best-first search with k look-ahead*, suffers from the same problem as the best-first search, it uses more information to make its decisions. The original best-first algorithm is the special

case of the look-ahead algorithm with $k=1$. The original best-first algorithm is modified as followed to implement this idea.

```

Best_First(queue,k)
queue: contains all the nodes in the slicing tree in
parent first order.
k: number of look-ahead
{
  Remove  $k$  elements from the head of the main queue
  and put them into Heu-queue.

  While (queue is not empty)
  {
     $P_{best} = A \text{ Big Number}$ 

    BandB(Heu_queue)

    Update  $P_{low}$  and net bounding boxes with
    the best relative position saved for the node
    at the head of the Heu_queue.

    Remove one element from the head of Heu_queue.

    Remove one element from the main queue and add it
    to the tail of Heu_queue.
  }
}

```

3.5. Summary

Up to this point, two different approaches to solve the wire length optimization problem have been presented. Although we know that the branch and bound algorithm solves the problem exactly, its run time might not be practical for typical circuits. On the other hand we know that the best-first search is too greedy in nature, but it may be quite efficient in terms of run time. The quality of the result from the best-first procedure must be investigated and compared with the exact solution from the branch and bound algorithm. The efficiency of these algorithms can be only tested by implementing them and applying them to practical VLSI circuits. The next chapter is devoted to a discussion of the implementation techniques and experimental results for the two algorithms.

Chapter Four

Implementation and Results

4.1. Preliminaries

The new algorithms presented in chapter 3, have been implemented in the C++ language on the UNIX operating system. C++ is an evolution of C which supports object-oriented design and programming. The new algorithms have been implemented using an object-oriented design methodology, and C++ was used because of its popularity among object-oriented languages.

Object-oriented design methodology represents an important view of software design. Object-oriented design views the construction of software systems as collections of *objects*. An object is a logical entity, like a real world component, which is mapped into the software domain. The software realization of an object contains both data structures and processes that manipulate the data structures. These processes are called *operations* or *methods*, and usually contain procedural and control constructs.

Since each object in an object-oriented design is a module,⁵ modularity is inherently provided in an object-oriented design. Modularity is an important desirable characteristic of software. A large program composed of a single module cannot be handled easily. The number of control paths, variables, and overall complexity of such a program make it difficult to understand and debug. Furthermore, modular software is flexible in architecture and achieves three important aspects of software quality: extendibility, reusability and compatibility.

⁵ The software is divided into separately named and addressable elements, called modules that are integrated to satisfy problem requirements. A simple form of a module can be a sub-routine or procedure, representing a step of the task to be performed by the software.

Extendibility is the ease with which software products may be adapted to changes of specifications. Reusability is the ability of software products to be reused for a new application, in whole or in part. Finally, compatibility is the ease with which software products may be combined with each other.

Another important issue in software design which is addressed by object-oriented design is *information hiding*. Information hiding is one of the key principles to ensure modularity, and implies that all information about a module should be private to the module and inaccessible to other modules that have no need for such information. The use of information hiding as a design criterion for modular programs provides its greatest benefits when modifications are required during testing and debugging. Besides, it has the advantage of providing a significant level of protection to the information contained within a module. Any accidental modification or incorrect usage of this information is prevented. Information hiding is achieved in an object-oriented design through private components of an object. Within an object, some of the data structures and/or methods can be private to the object and inaccessible to anything outside the object.

In describing objects in an object-oriented design, we are clearly more interested in classes of objects than in individual objects. *Class* is the technical term in object-oriented design to describe a set of data structures representing objects that are characterized by common properties. In order to have complete descriptions of the classes of data structures, we use *abstract data types*. In general, an abstract data type describes a class of data structures by the list of features available on the data structures and the formal properties of these features.

In order to implement an abstract data type, C++ has two constructs: the first is an extension of the *struct* construct in C, and the second is the *class* construct. The class construct is syntactically similar to struct. It has a name to identify it

and it can have data members and member functions (methods). Some or all of the members can be declared as private or public members. The public members are accessible both inside and outside the class, while the private members are only accessible within the class.

In object-oriented programming, the abstract data types are extended to allow for type/subtype relationships. This is achieved through a mechanism called *inheritance*. Rather than re-implementing shared characteristics, a class can inherit selected data members and member functions of other classes. In C++, inheritance is implemented through the mechanism of *class derivation*. A class that is inherited is referred to as a *base class* and the class that does the inheriting is called the *derived class*. When one class inherits another, the members of the base class become members of the derived class. In C++, the accessibility of the base class members inside the derived class can be controlled.

4.2. Software Overview

One program has been developed for each of the algorithms presented in the last chapter. Because of the similarities between these two algorithms, the programs have similar components, and the issues which are discussed here apply to both of them. The main body of the programs contains abstract data type definitions and procedures corresponding to the pseudo-code presented in the previous chapter. Besides these components, the software requires components to read the input files, initialize the data structures, and write the results in the output file. In this section, we first describe the abstract data types defined to implement the objects, then we briefly describe the input and output files, and finally we present the specifications of a program which produces a graphical representation of the algorithms' results.

4.2.1. Abstract Data Types

In order to provide an object-oriented architecture for the programs, the objects in the algorithms that we are trying to implement must be precisely identified. The objects which can be identified in the algorithms are from one of the following types:⁶ a node in the slicing tree, a net in the netlist of each node in the slicing tree, a module, a net in the netlist of each module, a net minimum bounding box and a node in the queue of slicing tree nodes. In order to implement each of the above objects, an abstract data type must be used. Each abstract data type describes the data members of each object and also the set of operations that may be applied to objects of that type. We use the C++ class construct to define the required abstract data types. The classes which have been defined to implement the above objects are as follows: 1) `basic_list`, 2) `list`, 3) `queue`, 4) `net`, 5) `module` and 6) `tree`. Each of these classes is explained in the following paragraphs.

The `basic_list` class

This class contains common features of the `queue` and `list` classes. The `list` class describes an element in a linked list and the `queue` class describes an element in a queue. There is a strong conceptual relationship between a queue and a linked list, the main difference being the operations which are performed on the two structures. Therefore, the `queue` and `list` classes can be defined as special cases of a single class which contains the common features between the two. Although the `basic_list` class does not describe any particular object in the program, it is a superclass of the two classes `queue` and `list`. The `basic_list` class specifications are as follows:

⁶ The final program contains other objects, but in this section we are concerned only with the objects required to implement the algorithms discussed in the previous chapter.

<i>Class:</i>	<i>basic_list</i>
<i>Superclass:</i>	<i>none</i>
<i>Private data,</i> <i>next:</i>	<i>pointer to an object of type basic_list.</i>
<i>Public Operations,</i> <i>constructor:</i>	<i>Initialize the object when it is created.</i>
<i>get_next():</i>	<i>Read the private data.</i>
<i>set_next(value):</i>	<i>Write the given value to the private data.</i>

This class is very simple and has only one private data member which is a pointer to another object of the same type. The pointer represents the concept of linkage between elements which is implicit in both the linked list and the queue data structures. This class also has three public member functions. The first function is a constructor which is invoked automatically each time an object of this type is created to initialize the data member. The other two functions (*get_next()* and *set_next()*) are used to access the private data, and anything outside the object can only access and modify the private data through using these functions.

The list class

The netlist for each node in the slicing tree and the netlist for each module are implemented as linked list data structures. Thus, defining a class for elements of a linked list seems to be sufficient to implement objects of both of these types. However, each object of one of these types has some unique attributes. For instance, a net in the netlist of a tree node contains *xmin()* and *ymin()* data elements, while a net in the module netlist contains *x* and *y* position of the net's terminals. In order to solve this problem a unique class is defined for elements in a linked list with some attributes, where each attribute has different interpretations for objects of either of the above types. This class is called *list*, which its specifications are as follows:

<i>Class:</i>	<i>list</i>
<i>Superclass:</i>	<i>basic_list</i>
<i>Private data,</i> <i>net_id:</i>	<i>net identifier.</i>
<i>x:</i>	<i>xmin() in the tree node netlist,</i> <i>x coordinate of the net terminal in the module netlist.</i>
<i>y:</i>	<i>ymin() in the tree node netlist,</i> <i>y coordinate of the net terminal in the module netlist.</i>
<i>w:</i>	<i>wmin() in the tree node netlist,</i> <i>terminal identifier in the module netlist.</i>
<i>h:</i>	<i>hmin() in the tree node netlist,</i> <i>not applicable in the module netlist.</i>
<i>Public Operations,</i> <i>constructor:</i>	<i>Initialize the object when it is created.</i>
<i>get_..():</i>	<i>Read the private data,</i> <i>available for each data member.</i>
<i>set_..(value):</i>	<i>Write the given value to the private data,</i> <i>available for each data member.</i>
<i>add_next():</i>	<i>Add a new element to the list and</i> <i>return a pointer to that.</i>
<i>find_bbox():</i>	<i>Find the position of partial net bounding box</i> <i>and the number of net terminals within a module,</i> <i>applicable only in the module netlist.</i>

As illustrated above, the class has a couple of private data elements, which have different interpretations for the netlist of a node in the slicing tree and the netlist of a module. This class has different member functions, such as a constructor, the functions for accessing to the private data members, and a function for adding new elements to an existing list. The other member function which needs explanation is *find_bbox()*. As stated in the previous chapter, the netlist for a module or a node in the slicing tree is sorted based on the net id-numbers. When this operation is applied to an element in the list, it goes through the next elements of the list which are objects of the same type and compares the position of terminals until it reaches an element with a different net id-number. Then it returns the position of net bounding box and the number of terminals.

The queue class

Like the class list, the queue class is also derived from the class `basic_list`. The only private data in this class is a pointer to an object of the class tree, which will be discussed later. This class specifications appears below:

<i>Class:</i>	<i>queue</i>
<i>Superclass:</i>	<i>basic_list</i>
<i>Private data,</i> <i>node:</i>	<i>pointer to a node in the slicing tree.</i>
<i>Public Operations,</i> <i>constructor:</i>	<i>Initialize the object when it is created.</i>
<i>get_node():</i>	<i>Read the private data.</i>
<i>set_node(value):</i>	<i>Write the given value to the private data.</i>
<i>add(head,tail):</i>	<i>Add a new element to the tail of the queue, and return a pointer to that.</i>
<i>remove(head,tail):</i>	<i>Remove an element from the head of the queue, and return a pointer to the new head.</i>
<i>undo(head,tail):</i>	<i>Add an element to the head of the queue, and return a pointer to the new head.</i>

Besides the public member functions for accessing the private data, there are three more operations that can be applied to objects of this class. Each of these operations has two arguments which specifies the tail and head of a queue data structure. Except for `remove()`, which is only applied to the head of the existing queue, the other two operations can be applied to any object of this type. When `add()` or `undo()` is applied to an object, that object becomes part of the existing queue. For all of these operations, the head or tail of the queue must be updated to the new values returned by these operations.

The net class

This class is defined to describe net minimum bounding boxes. It contains private data members to represent the dimensions of minimum bounding boxes and the total number of net terminals. The specifications of the net class appears

below:

<i>Class:</i>	<i>net</i>
<i>Superclass:</i>	<i>none</i>
<i>Private data,</i>	
<i>r:</i>	<i>minimum bounding box right coordinate.</i>
<i>l:</i>	<i>minimum bounding box left coordinate.</i>
<i>t:</i>	<i>minimum bounding box top coordinate.</i>
<i>b:</i>	<i>minimum bounding box bottom coordinate.</i>
<i>w:</i>	<i>minimum width of net overall bounding box.</i>
<i>h:</i>	<i>minimum height of net overall bounding box.</i>
<i>total:</i>	<i>total number of net terminals.</i>
<i>Public Operations,</i>	
<i>constructor:</i>	<i>Initialize the object when it is created.</i>
<i>get_..():</i>	<i>Read the private data, available for each data member.</i>
<i>set_..(value):</i>	<i>Write the given value to the private data, available for each data member.</i>
<i>store():</i>	<i>Save the bounding box coordinates,</i>
<i>restore():</i>	<i>Change the bounding box coordinates according to the previously saved values.</i>

Among the private data members in this class, *w* and *h* are the *net_width()* and *net_height()* parameters explained in the previous chapter. As can be seen from the class specification, there is no net id-number among the private data members. In order to describe nets with different id-numbers, an array of pointers is defined where each element of this array points to an object of this class. The index of that element of the array is the id-number of the net which is described by the object pointed to.

The module class

This class is defined to describe the circuit modules. This class is also fairly simple and is specified as follows:

<i>Class:</i>	<i>module</i>
<i>Superclass:</i>	<i>none</i>

<i>Private data,</i>	
<i>x:</i>	<i>x coordinate of left bottom corner of the module.</i>
<i>y:</i>	<i>y coordinate of left bottom corner of the module.</i>
<i>width:</i>	<i>width of the module.</i>
<i>height:</i>	<i>height of the module.</i>
<i>net_list:</i>	<i>pointer to the beginning of module netlist.</i>

<i>Public Operations,</i>	
<i>constructor:</i>	<i>Initialize the object when it is created</i>
<i>get_..():</i>	<i>Read the private data, available for each data member.</i>
<i>set_..(value):</i>	<i>Write the given value to the private data, available for each data member.</i>

Again an array of pointers to objects of this class is defined. The index of each element of this array is id-number of the module which is described by the pointed object of this class.

The tree class

This class which is the most important class in our programs, is defined to describe a node (internal or leaf) in the slicing tree structure. The specifications of this class appears below:

<i>Class:</i>	<i>tree</i>
<i>Superclass:</i>	<i>none</i>
<i>Private data,</i>	
<i>a:</i>	<i>relative position assigned to the node.</i>
<i>besta:</i>	<i>best value of "a" seen up to the current point in the search.</i>
<i>c:</i>	<i>slice direction assigned to a node.</i>
<i>x, y:</i>	<i>x & y coordinates of the left bottom corner of the rectangle corresponding to the node.</i>
<i>slice:</i>	<i>x or y coordinates of the slice line assigned to the internal node depending on the slice direction, not applicable to leaf nodes.</i>
<i>width, height:</i>	<i>width & height of the rectangle corresponding to the node.</i>
<i>nets:</i>	<i>pointer to the beginning of the node netlist (List()).</i>
<i>module:</i>	<i>id-number of the module assigned to a leaf node, not applicable to internal nodes.</i>
<i>left:</i>	<i>pointer to left child node.</i>
<i>right:</i>	<i>pointer to right child node.</i>

Public Operations,

<i>constructor:</i>	<i>Initialize the object when it is created.</i>
<i>get_..():</i>	<i>Read the private data, available for each data members.</i>
<i>set_..(value):</i>	<i>Write the given value to the private data, available for each data member.</i>
<i>is_leaf():</i>	<i>Return 1 when the object to which it is applied corresponds to a leaf node.</i>
<i>has_ver_slice():</i>	<i>Return 1 when the object to which it is applied corresponds to an internal node with vertical slice.</i>
<i>save_best():</i>	<i>Write "a" to "besta"</i>
<i>slice_cal():</i>	<i>Calculate the position of the slice line for an internal node</i>
<i>update_node():</i>	<i>Update P_{low} and the minimum bounding box boundaries for the nets in List(), applicable only to internal nodes.</i>
<i>update_leaf():</i>	<i>Update P_{low} and the minimum bounding box boundaries for the nets in List(), applicable only to leaf nodes.</i>
<i>add_left():</i>	<i>Add a left child to the node to which it is applied and return a pointer to that node.</i>
<i>add_right():</i>	<i>Add a right child to the node to which it is applied and return a pointer to that node.</i>

Each object of this class has many data members. Among the various data members, there are two pointers to the objects of the same type. One of them points to the node's left child and the other points to the node's right child. Each object has also a pointer to an object of class list which describes the node netlist. Among the member functions, two of the operations, *update_node()* and *update_leaf()* are the most important operations in this class. The pseudo-code of these operations was presented in the previous chapter.

4.2.2. Input and Output Files

The program for each of the algorithms requires two input files and the results are written to a single output file. The parser for the input files first checks the syntax of input files and reports any errors. If there are no errors, the input files are read again and used to establish the required data structures.

One of the input files contains the module descriptions and the circuit netlist. This file is expected to have the extension of *.cel* in its name. An example of a typical module, named C-5, described in a *.cel* file is as follows:

```
cell C-5;
width 450;
height 300;
net 87 x 210 y 63 term 0;
net PGI x 340 y 22 term 1;
net VDD x 40 y 280 term 2::
```

In the *.cel* file *cell*, *width*, *height*, *net*, *x*, *y* and *term* are the reserved keywords. The string of characters following the keywords *cell* and *net* are the names assigned to a module and a net. These names can be any combination of alphanumeric characters, underscore or minus sign. The rest of the data elements are numbers. The dimensions of each module, in addition to all of the attached nets and coordinates of the terminals with respect to the left bottom-corner of the module are available in the *.cel* file. All of the dimensions and coordinates in this file are in microns.

```
1 V;
2 H;
3 L C-1;
4 L C-2;
5 H;
6 L C-3;
7 L C-4;
```

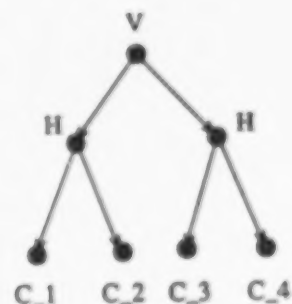


Figure 4-1: Description of *.tre* file

The second input file contains the structure of the slicing tree. This file is expected to have the extension of *.tre* in its name. Each row in this file corresponds to a node in the slicing tree and the node descriptions appear in depth first order. The *.tre* file format which describes a simple slicing tree with its

corresponding slicing tree is illustrated in Figure 4-1. In the *.tre* file *H*, *V* and *L* are the reserved keywords. *H* and *V* represent horizontal and vertical slice directions assigned to an internal node. *L* indicates that the node is a leaf node and the name of the module assigned to that leaf node follows the *L*.

The output file, which has *.out* extension, is basically the same as the *.tre* file. In addition to all of the information in the *.tre* file, it contains the relative positions assigned to each node in the slicing tree. An instance of a *.out* file for the example of Figure 4-1 follows:

```

1  V  F;
2  H  R;
3  L  C-0  RU;
4  L  C-1  FR;
5  H  F;
6  L  C-2  RR;
7  L  C-3  FU;

```

4.2.3. Graphical Representation of the Floorplans

In order to understand the results output by the programs, it is essential to have a graphical representation of the floorplans before and after running the programs. If in addition to the location of the modules, the net bounding boxes are also illustrated, then we can have a better view of the effect the algorithms have on compacting the net bounding boxes.

X windows is an ideal environment for generating such a graphic representation. It is a software environment for engineering workstations which has become accepted as a standard graphical interface to the workstations. X windows offers a large number of graphic capabilities. Some of the capabilities which are related to our requirements are as follows:

- 1) X windows organizes display screens into a hierarchy of overlapping windows. Each application can use as many windows as it needs, resizing,

moving and stacking them on top of one another as needed.

- 2) X windows provides drawing capabilities. X's graphics operations are immediate rather than display list oriented. In other words, the workstation does not save series of graphics operations, but rather draws everything immediately.
- 3) X windows drawing operations are bit mapped. Each application specifies all the operations in terms of integer pixel addresses within a window. All graphics operations are addressed within a particular window, so applications can draw things in their window without regard to where their windows are positioned on the screen.
- 4) X windows supports drawing high quality text for a wide range of applications.

The program which is developed to illustrate the floorplans and the net bounding boxes in the X windows environment is written in C. A C language subroutine package known as *Xlib* is provided within the X windows system. *Xlib* makes programming much easier and the programmer can avoid dealing with the internal complexities of the X windows system. The floorplan drawing program opens two equal sized windows in an arbitrary position on the screen, one for the floorplan before running the algorithms and one for the floorplan after running the algorithms. The modules are drawn in each window and the net bounding boxes can be drawn optionally. All drawing is done through a series calls to subroutines in *Xlib*. Everything drawn to a window will be scaled up or down to fit in the window. Therefore two illustrations of one floorplan before and after running the algorithms have the same scales, whereas the illustrations for two different floorplans may not.

4.3. Experimental Results

The efficiency of the presented algorithms was evaluated by applying them to practical VLSI circuits. Five standard benchmark circuits, which are provided by the Microelectronic Center of North Carolina (MCNC), were used as example circuits. The number of modules, nets and terminals in each of these circuits is presented in Table 4-1.

Benchmark	Number of modules	Number of nets	Number of terminals
Xerox	10	203	796
Apte	9	40	214
Hp	11	56	264
Ami33	33	89	480
Ami49	49	408	931

Table 4-1: Statistics for the example circuits

To run the algorithms for a given circuit, two input files are required. The first one, the *.cel* file, can be easily prepared from the available netlist and cell information for each of these circuits. The second file, the *.tre* file, must be obtained from a given floorplan. For the evaluation, we prepared the *.tre* file from the layout resulting from another algorithm published by Onodera *et al.* That algorithm does not necessarily produce slicing floorplans, thus the published layouts are slightly modified to provide a slicing floorplan. The relative module locations were copied exactly from the layouts provided in the paper; however, the module orientations could not be determined from the published layouts and these were arbitrarily defined.

Two measures are important in order to evaluate the efficiency of the algorithms. The first is the running time of the algorithms, which was measured through the *time* command in the Unix system. The time command gives different time measures for each process. Among these measures we are interested in "user time"⁷ which is measured in seconds and denoted by T .

The second parameter which is important is the total half perimeter of the circuit. This parameter tells us how effective the algorithms are in compacting the net bounding boxes. We denote the total half perimeter for the floorplans before and after running the algorithms by P_{init} and P_{fin} respectively.

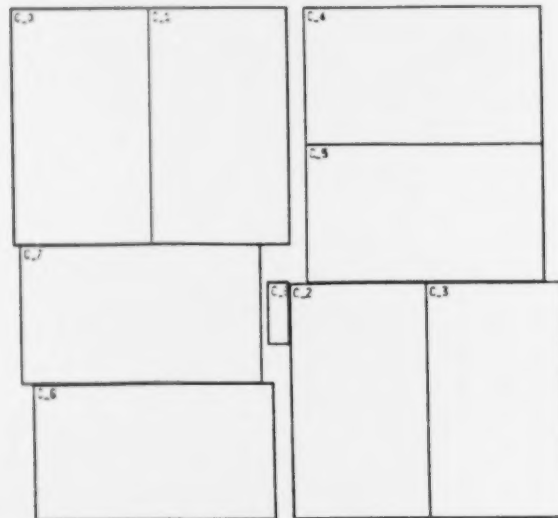
Benchmark	P_{init}	Depth First			Breadth First		
		P_{fin}	%	T	P_{fin}	%	T
Xerox	413511	395255	4	131.0	395255	4	639.8
Apte	250497	160258	36	50.3	160258	36	8.4
Hp	159827	116101	27	858.5	116101	27	4251.2

Table 4-2: The results of the branch and bound algorithm

The results of applying the branch and bound algorithm to the benchmarks are presented in Table 4-2. As can be seen, the results for *Ami33* and *Ami49* are not presented in this table. The branch and bound algorithm did not terminate for these circuits within a practical length of time. For each benchmark in the table, T , P_{init} , P_{fin} and the improvement percentage of P_{fin} over P_{init} are given.

⁷ The Unix system allows two different modes of operation: user mode and system mode. In the user mode the execution is done on behalf of the user, while in system mode the operating system gains the control of the computer. Every process in Unix has both a user and system phase. The user time refers to the amount of time that the computer spends for a process in the user mode.

a) Before running the algorithm



b) After running the algorithm

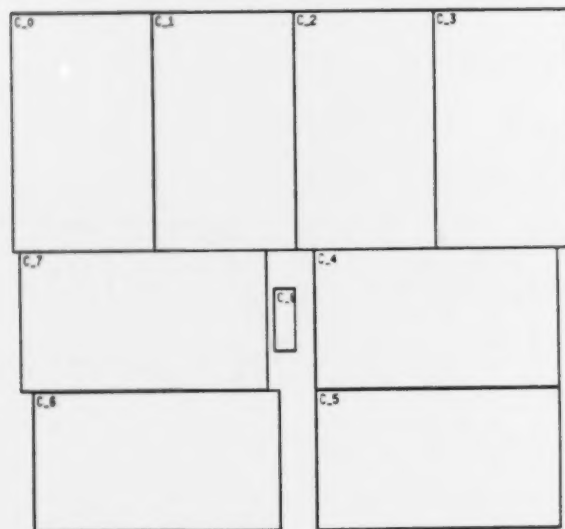
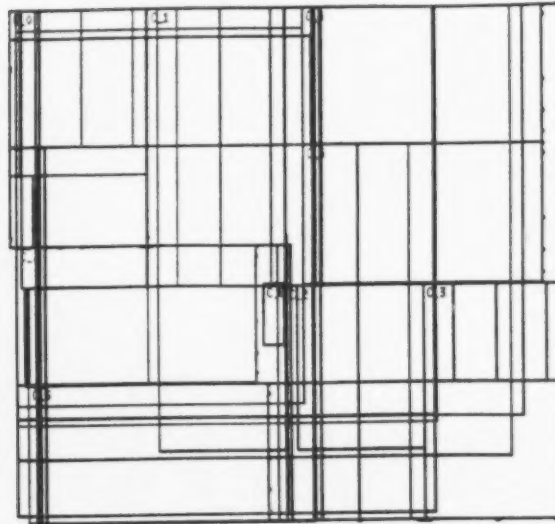


Figure 4-2: Apte module locations

a) Before running the algorithm



b) After running the algorithm

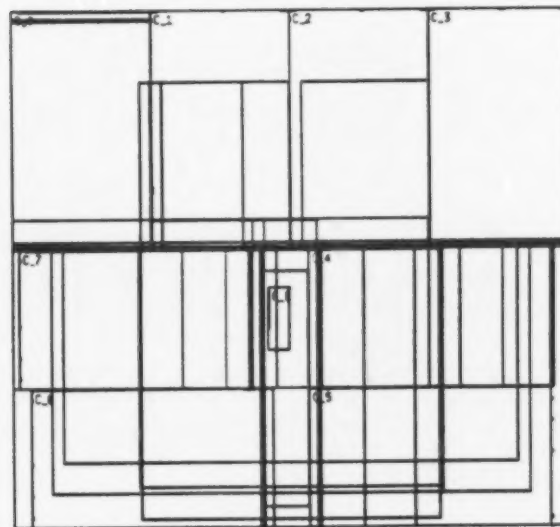


Figure 4-3: Apte module locations and net bounding boxes

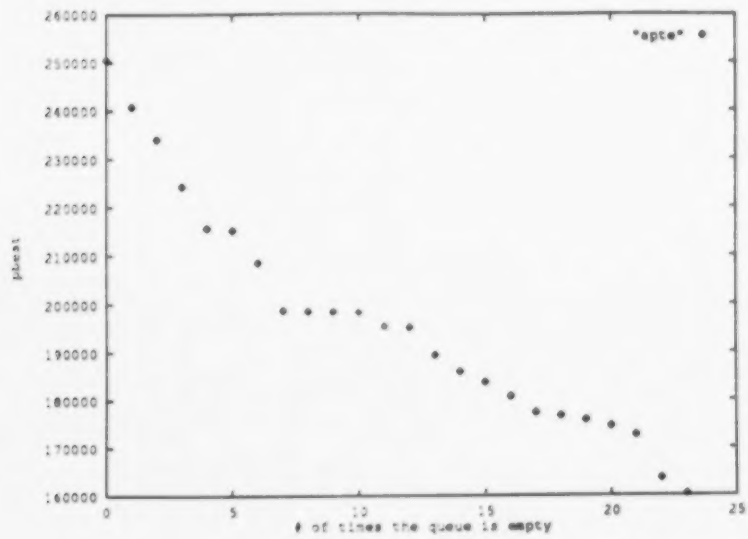
As can be seen from Table 4-2, the branch and bound algorithm was able to improve each of the given circuits from 4% to 36%. The improvement made in these circuits is not only due to mirroring the cells in the Onodera layouts, but also because of some changes in the location of the modules. The initial and final floorplans for the *Apte* circuit are illustrated in Figure 4-2. Furthermore, the net bounding boxes for the *Apte* initial and final floorplans are illustrated in Figure 4-3. The significant effect of the branch and bound algorithm in compacting net bounding boxes is obvious from this figure. The initial and final floorplans and net bounding boxes for the other two circuits, *Xerox* and *Hp* are presented in the Appendix.

The branch and bound algorithm for each benchmark was tried for two different orders of the nodes in the slicing tree: depth first and breadth first. Although the final floorplan resulting from these two orders is the same, the run time is quite different. For instance for *Hp* the run time for the breadth first order is almost five times greater than the time for the depth first order. In contrast, for *Apte* the run time for the depth first order is almost six times larger than the time for breadth first order. There is no general trend which describes whether one of these orders is superior for a given problem.

An interesting issue in the branch and bound algorithm is to see how the value of P_{best} changes during the run time. The plots of changes in the value of P_{best} in *Apte* for the depth first order is illustrated in Figure 4-4. The first plot in this figure gives P_{best} versus the number of times the queue becomes empty. As can be seen from this plot, the changes in P_{best} happen smoothly. The second plot illustrates the P_{best} changes versus time. This plot demonstrates that changes in P_{best} versus time is not smooth. This is due to the fact that the value of P_{best} is changed only when the queue becomes empty and the time required for the

queue to become empty is random.

a) P_{best} versus number of times the queue is empty



b) P_{best} versus time

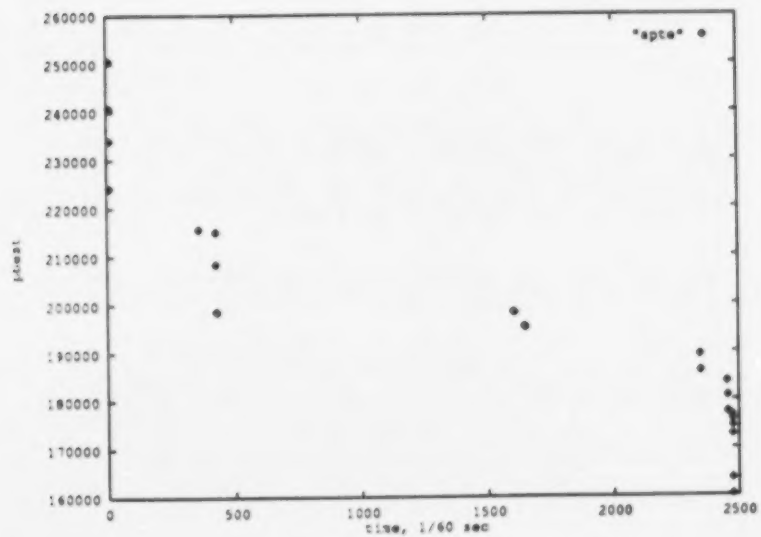
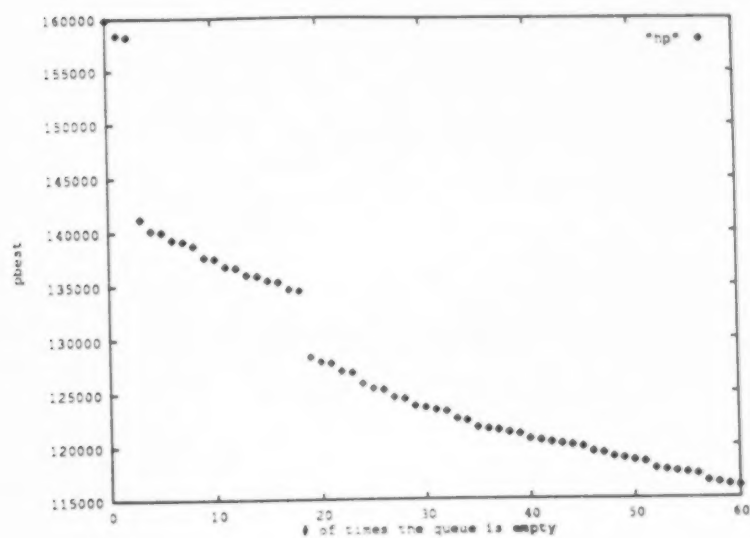


Figure 4-4: Plots of P_{best} changes in Apte

a) P_{best} versus number of times the queue is empty



b) P_{best} versus time

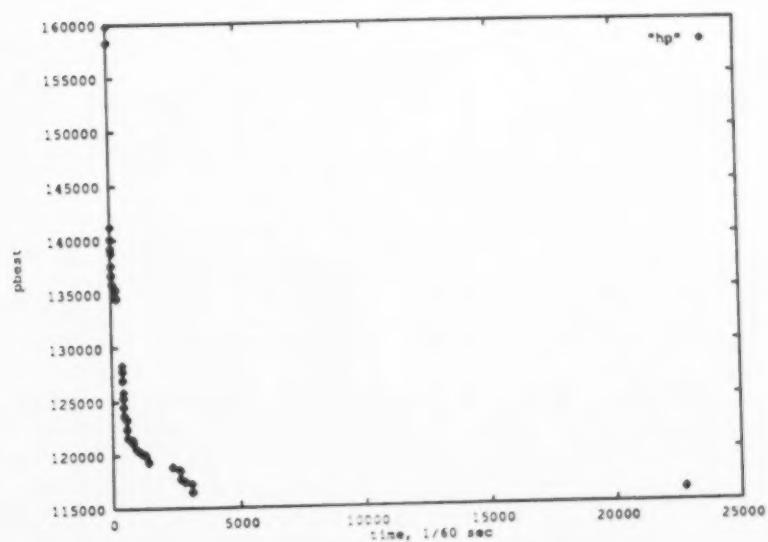


Figure 4-5: Plots of P_{best} changes in Hp

The plots of P_{best} in Hp for the depth first order are illustrated in Figure 4-5. There is an interesting situation in this figure. Except for two big jumps, the changes in P_{best} are smooth and most of the changes are made during the first one-fifth of the run time. In the rest of the time there is no greater improvement in P_{best} . In other words, by setting a time limit on the algorithm we can get a good solution for the Hp problem more rapidly. The plots of P_{best} in $Xerox$ are given in Appendix.

Another interesting issue in the branch and bound algorithm is to see how the value of P_{low} approaches the actual total half perimeter of circuit. P_{low} is the lower bound on the total half perimeter of the circuit and the initial value of P_{low} is an unrealistic value for the total half perimeter of the circuit. The plot of changes in P_{low} during the first time that the queue becomes empty is illustrated in Figure 4-6. As can be seen in this figure, the initial value of P_{low} is far from the final value, which is the actual total half perimeter of the circuit. Similar plots of P_{low} in $Xerox$ and Hp are given in the Appendix.

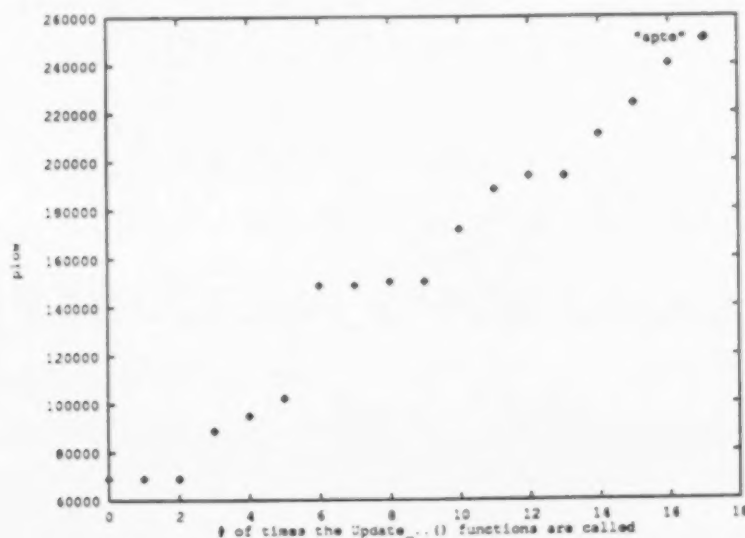


Figure 4-6: Plot of P_{low} versus the number of times the Update_..() functions are called, in Apte

By considering the results of the branch and bound algorithm in Table 4-2, it can be seen that in some cases the run time is very large. In order to improve the speed of the algorithm, we can let the algorithm search for reasonably good solutions instead of for an optimum solution. When the objective of the search is sub-optimal solutions, the branches in the state space tree are cut more rapidly. In order to implement this idea, the branch and bound algorithm must be modified slightly. Instead of comparing P_{low} with P_{best} , the value of $(1 + \text{desired percentage}) \times P_{low}$ is compared against P_{best} . The *desired percentage* is the margin by which the produced solutions can exceed the optimum solution. The results of this modification to the branch and bound algorithm, where the margin was set to 5 percent, are presented in Table 4-3.

Benchmark	P_{init}	Depth First			Breadth First		
		P_{fin}	%	T	P_{fin}	%	T
Xerox	413511	413511	0	108.4	413511	0	539.1
Apte	250497	163640	35	33.2	163640	35	6.0
Hp	159827	121695	24	798.3	116241	27	162.7

Table 4-3: The results of the branch and bound algorithm for 5% solutions

By examining entries in this table and comparing them with Table 4-2, it is easily realized that this modification decreases the run time, but not significantly.

The best-first algorithm was also evaluated experimentally. The best-first algorithm can be applied with a different number of look-ahead(k). In Table 4-4, the results of applying the basic best-first algorithm with $k=1$ for depth first and breadth first node orders are presented.

Benchmark	P_{init}	Depth First			Breadth First		
		P_{fin}	%	T	P_{fin}	%	T
Xerox	413511	403081	2	1.1	404047	2	1.0
Apte	250497	163950	35	0.3	166462	34	0.3
Hp	159827	122074	24	0.3	123472	23	0.3
Ami33	54512	37385	31	0.6	37603	31	0.6
Ami49	873824	731186	16	2.2	723416	17	2.0

Table 4-4: The results of the best-first algorithm with $k=1$.

Comparing the results in this table with the results from the branch and bound algorithm in Table 4-2, shows the best-first algorithm to be very effective. All of the results were achieved very quickly including *Ami33* and *Ami49*. Note that the results from the two different ordering of the nodes are very close to each other.

Benchmark	P_{init}	Depth First			Breadth First		
		P_{fin}	%	T	P_{fin}	%	T
Xerox	413511	403081	2	1.3	404047	3	1.2
Apte	250497	163950	35	0.4	164072	35	0.4
Hp	159827	116446	27	0.5	117844	26	0.5
Ami33	54512	37333	32	0.8	37319	32	0.8
Ami49	873824	724628	17	2.7	714171	18	2.6

Table 4-5: The results of the best-first algorithm with $k=2$.

Our hope is that better results can be achieved by increasing the amount of look-ahead. In Table 4-5, the results of applying the best-first algorithm with $k=2$,

again for depth first and breadth first orderings, are presented. The results for *Xerox* are the same as the results for $k=1$. For *Apte* the result for breadth first order is slightly improved. For *Hp* we have almost 3% improvement over the results for $k=1$. For *Ami33* the solution is slightly worse than the one for $k=1$. Finally, for *Ami49* we had a slight improvement. In general, we can expect improvement by increasing k , but because the algorithm is not optimum, the situations like in *Ami33* might arise.

Benchmark	P_{ini}	Depth First			Breadth First		
		P_{fin}	%	T	P_{fin}	%	T
Xerox	413511	395255	4	1.8	404047	3	1.8
Apte	250497	163950	35	0.6	164072	35	0.6
Hp	159827	116101	27	0.7	117844	26	0.7
Ami33	54512	37319	32	1.3	37319	32	1.2
Ami49	873824	723274	17	4.0	719854	18	4.0

Table 4-6: The results of the best-first algorithm with $k=3$

In Table 4-6 and 4-7, the results of applying the algorithm for $k=3$ and $k=5$ are presented. For *Xerox* and *Hp* with $k=3$ and $k=5$ the optimum solution is obtained. For *Apte*, by increasing k the optimum solution could not be achieved. For *Ami33* and *Ami49* the results are generally improved by increasing the value of k . The initial and final floorplans and net bounding boxes for *Ami33* and *Ami49* resulting from the algorithm with $k=5$ and depth first ordering are presented in Appendix.

Benchmark	P_{init}	Depth First			Breadth First		
		P_{fin}	%	T	P_{fin}	%	T
Xerox	413511	395255	4	5.0	402402	3	5.8
Apte	250497	163950	35	1.7	164072	35	1.3
Hp	159827	116101	27	2.3	117221	27	2.1
Ami33	54512	37030	32	4.1	37859	31	4.5
Ami49	873824	714583	18	13.9	719854	18	13.3

Table 4-7: The results of the best-first algorithm with $k=5$

In general, the best-first algorithm is quite efficient and, in some cases, is even able to achieve the optimum solution. Furthermore, it made a significant improvement in *Ami33* and *Ami49* where the branch and bound fails to finish after hours. It should be mentioned that, for all the circuits, the improvement made by the algorithm is not only due to mirroring the cells in the initial floor-plans, but also because of some changes in the location of the modules.

Chapter Five

Conclusions

In this thesis we considered the problem of wiring optimization in slicing floorplans. The problem which was addressed was how to re-arrange the modules in a given slicing floorplan so as to minimize total wire length of the circuit, while leaving the overall shape and area of the floorplan unchanged. In order to clearly specify this problem, the slicing tree representation of a slicing floorplan was used. For each internal node in the slicing tree, there are two possible relative positions of its children, and for each leaf node there are four possible orientations of the module assigned to the leaf. The problem was to find the relative positions of all the internal nodes, and the orientations of the modules assigned to the leaf nodes that minimize total wire length. As an estimate of total wire length, we used the total half perimeter of the net bounding boxes.

This approach to minimizing the total wire length of a floorplan has not been attempted previously. In existing algorithms for wiring optimization, wire length is just one component of a cost function, along with chip area, and this results in a trade-off between minimum area and minimum total wire length. In our approach, the objective was to minimize the total wire length while not changing the chip area.

To solve the wire length optimization problem, two algorithms were presented. The first one is a branch and bound algorithm which operates by keeping the lower bound on the total half perimeter of the circuit. This algorithm visits nodes in the slicing tree in parent first order. At each node, it assigns a relative position to the node and updates P_{low} . Then the value of P_{low} is compared with P_{best} , and if P_{low} is less than P_{best} , the algorithm continues the search based

on the assigned relative position. The algorithm also tries the alternate relative position for that node and repeats the procedure.

The second algorithm presented in this thesis was a greedy algorithm called the best-first algorithm. The best-first algorithm is based on ideas and techniques from the branch and bound algorithm. It visits nodes in the slicing tree in parent-first order and it assigns to each node the relative position which results in the lowest P_{low} .

The presented algorithms were implemented in the C++ language within the UNIX operating system using an object oriented methodology. The efficiency of the algorithms was evaluated by applying them to benchmark circuits provided by Microelectronic Center of North Carolina (MCNC). The results showed that the branch and bound algorithm is only practical for small floorplans. The most important parameter which affects the run time of the algorithm is the number of modules in the floorplan, and the algorithm is impractical for floorplans with 15 modules or more.

The best-first algorithm is quite efficient in terms of run time. For small circuits, for which branch and bound is practical, the comparison between the results of the branch and bound and the best-first demonstrates that, most of the time, the best-first algorithm is able to achieve solutions within 5% of the optimum solution, and even yields the optimum solution occasionally. For larger floorplans, the best-first algorithm is able to obtain good solutions, with considerable improvement over the initial floorplans, very quickly.

Both of the algorithms presented in this thesis have practical applications. Prior to the wiring of a floorplan, these algorithms can be applied to improve the wiring quality of the layout. When the number of modules in the floorplan is small, the branch and bound can be applied; when the number of modules are

higher, the best-first algorithm can be employed. The results of applying the algorithms to the benchmark circuits show that there is still room for minimizing the total wire length of floorplans produced by the other floorplanning algorithms. This point proves that our approach can be employed in the existing floorplanning systems. The other issue which should be noted is that, although minimizing the total half perimeter of the circuit does not directly reduce the wiring area or wiring delay, it can be expected that it does make an overall improvement in both of these measures.

Future work could include finding better heuristics for the wire length minimization problem. The results of applying the best-first algorithm are encouraging, but it is not an optimum algorithm. Unfortunately, heuristic searches have inherent limitations. Since they use limited information, they are seldom able to predict the exact behavior of the problem state space farther along in the search. Therefore, expecting to have an exact heuristic is unrealistic, but we can still hope to have more efficient heuristics that can predict the problem behavior better.

Another issue which can be considered in the future is to generalize the problem to non-slicing floorplans. The algorithms presented here are only applicable to slicing floorplans, but it may be possible to extend these algorithms to more general floorplans. For instance, *pinwheels*, which are the simplest non-slicing floorplans, could perhaps be handled this way.

The most important issue which should be the subject of future work is to consider the timing performance of the floorplan during the re-arranging of modules. The constraints on the delays of critical paths in the circuit should be considered when the relative positions of the nodes in the slicing tree are determined. The decision on the relative positions would be made so as not to violate the delay constraints of the critical paths. By implementing such an algorithm, an

existing floorplan could be modified to achieve better timing performance.

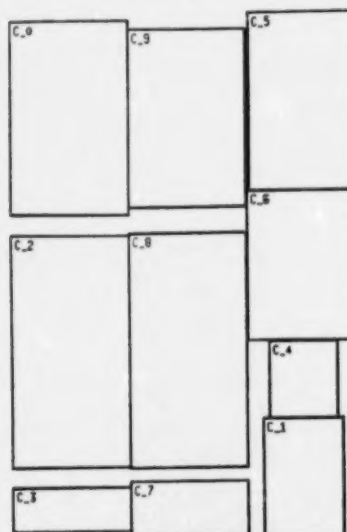
References

1. H. Anway, G. Farnham, and R. Reid, "Plint Layout System for VLSI Chips," *Proc. 22nd Design Automation Conference*, pp. 449-452, 1985.
2. S. P. Bradley, A. C. Hax, and T. L. Magnanti, "Chapter 2: Solving Linear Programs," in *Applied Mathematical Programming*, pp. 48-90, Addison Wesley, 1977.
3. M. A. B. Jackson and E. S. Kuh, "Performance-driven Placement of Cell Based ICs," *Proc. 26th Design Automation Conference*, pp. 370-375, 1989.
4. D. Jepsen and D. Gelatt, "Macro Placement by Monte Carlo Annealing," *Proc. 1983 IEEE International Conference on Computer Design*, pp. 495-498, November 1983.
5. S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi, "Optimization by Simulated Annealing," *Science*, vol. 220, no. 4598, pp. 671-680, May 1983.
6. D. P. Lapotin and S. W. Director, "Mason: A global floorplanning approach for VLSI design," *IEEE Trans. on CAD of ICs and Systems*, vol. CAD-5, no. 4, pp. 477-489, October 1986.
7. B. Lokanathan and E. Kinnen, "Performance Optimized Floorplanning by Graph Planarization," *Proc. 26th Design Automation Conference*, pp. 116-121, 1989.
8. R. Otten, "Layout Compilation," in *Design Systems for VLSI circuits, Logic Synthesis and Silicon Compilation*, ed. P. Antognetti, pp. 113-195, Martinus Nijhoff Publishers, 1987.
9. S. Prasitjutrakul and W. J. Kubitz, "Path-Delay Constrained Floorplanning: A Mathematical Programming Approach for Initial Placement," *Proc. 26th Design Automation Conference*, pp. 364-369, 1989.

10. M. Rose, M. Wiesel, D. Kirkpatrick, and N. Nettleton, "Dense, Performance Directed, Auto Place and Route," *Proc. IEEE Custom Integrated Circuit Conference*, pp. 11.1.1-11.1.4, 1988.
11. C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE Journal of Solid State Circuits*, vol. sc-20, no. 2, pp. 510-522, April 1985.
12. L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs," *Information and Control*, vol. 59, pp. 91-101, 1983.
13. K. Ueda, H. Kitazawa, and I. Harada, "CHAMP: Chip Floorplan for Hierarchical VLSI Layout Design," *IEEE Trans. on CAD of ICs and Systems*, vol. CAD-4, no. 1, pp. 12-22, January 1985.
14. M. P. Vecchi and S. Kirkpatrick, "Global Wiring by Simulated Annealing," *IEEE Trans. on CAD of ICs and Systems*, vol. CAD-2, no. 4, pp. 215-222, October 1983.
15. D. F. Wong and C. L. Liu, "A new algorithm for floor plan design," *Proc. 23rd Design Automation Conference*, pp. 101-107, 1986.
16. A. R. Newton and A. L. Sangiovanni-Vincentelli, "Computer-Aided Design for VLSI Circuits," *IEEE Computer*, vol. 19, no. 4, pp. 38-63, April 1986.
17. R. Otten, "Automatic Floor-plan Design," *Proc. 19th Design Automation Conference*, pp. 261-267, 1982.
18. H. Onodera, Y. Taniguchi, and K. Tamaru, "Branch-and-bound Placement for Building Block Layout," *Proc. 28th Design Automation Conference*, pp. 433-439, 1991.

Appendix

a) Before running the algorithm.



b) After running the algorithm.

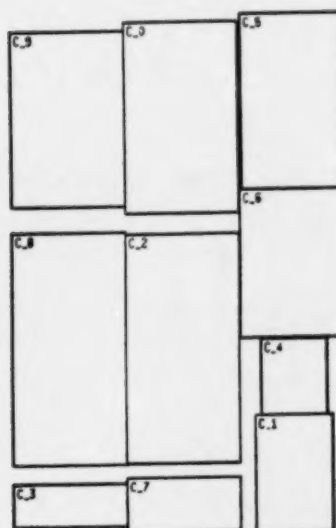
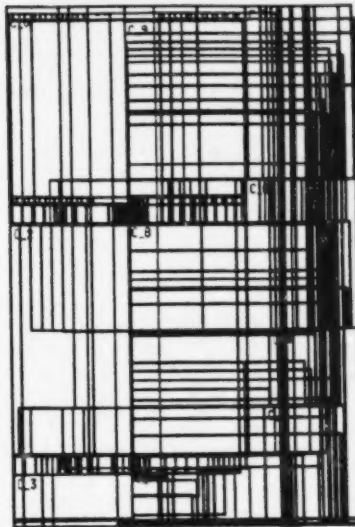


Figure A-1: Xerox module locations, produced by the branch and bound algorithm.

a) Before running the algorithm.



b) After running the algorithm.

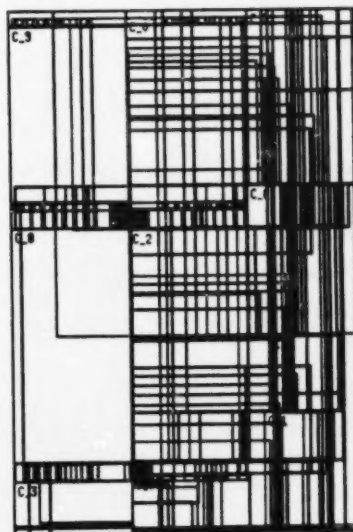
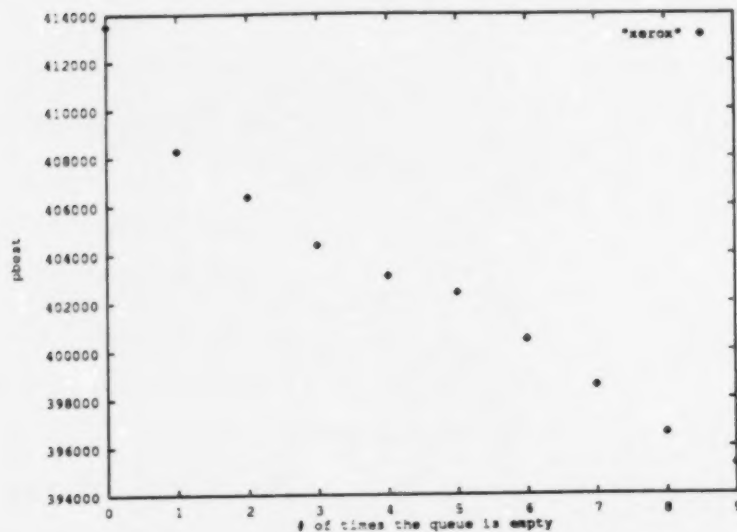


Figure A-2: Xerox module locations and net bounding boxes, produced by the branch and bound algorithm.

a) P_{best} versus number of times the queue is empty



b) P_{best} versus time

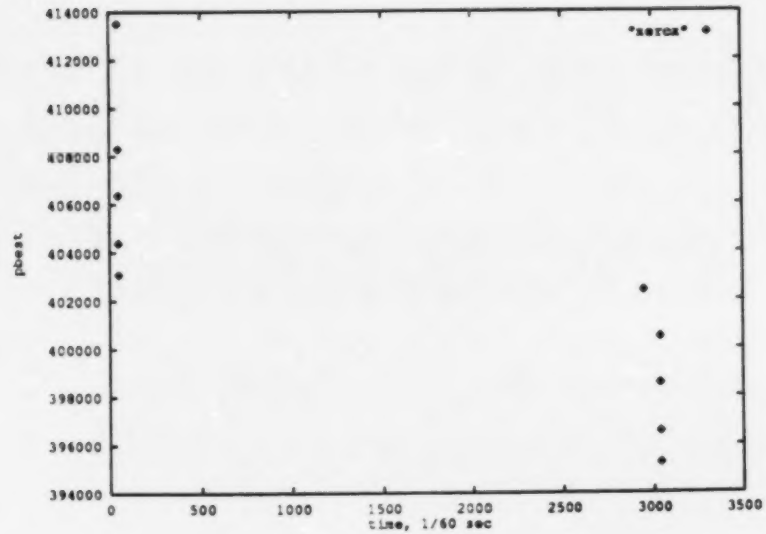


Figure A-3: Plots of P_{best} changes in Xerox.

a) Before running the algorithm.



b) After running the algorithm.

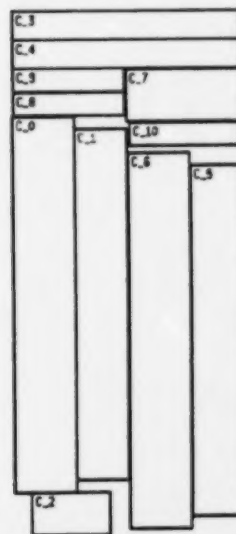
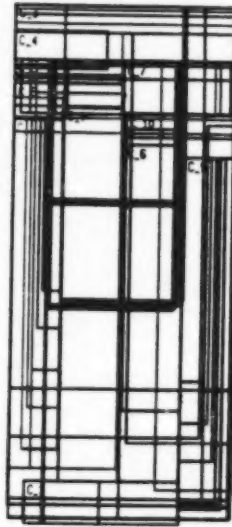


Figure A-4: Hp module locations, produced by the branch and bound algorithm.

a) Before running the algorithm.



b) After running the algorithm.

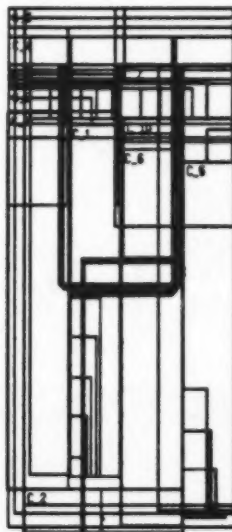


Figure A-5: Hp module locations and net bounding boxes, produced by the branch and bound algorithm.

Figure A-6: Plot of P_{low} versus the number of times the Update_..() functions are called, in Xerox.

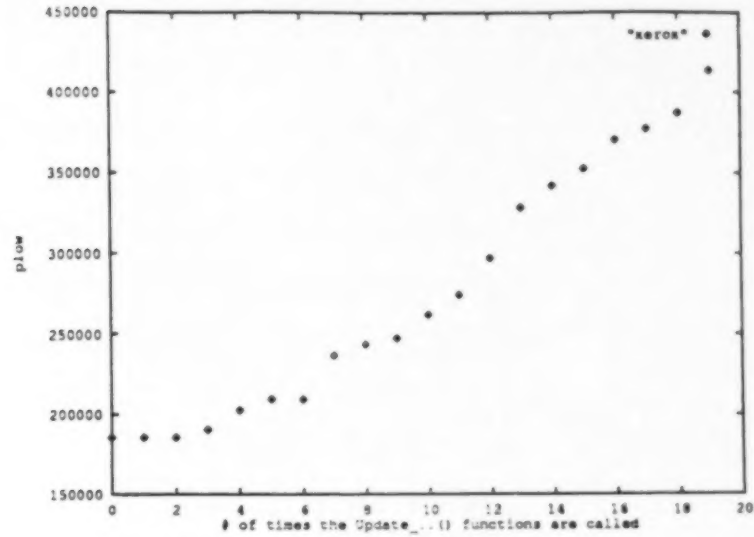
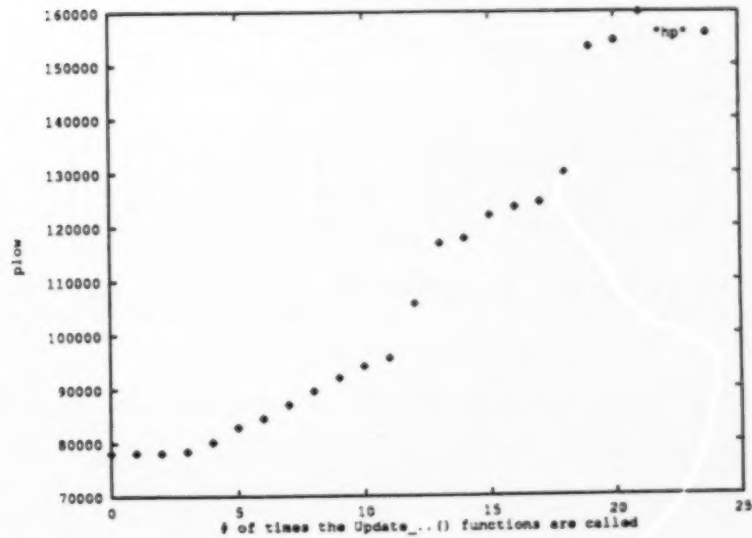
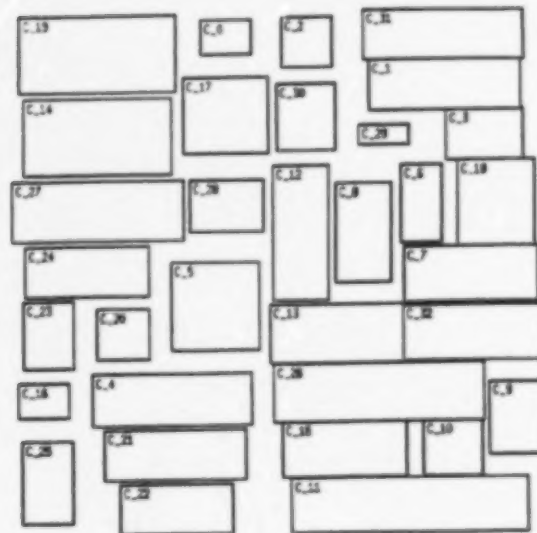


Figure A-7: Plot of P_{low} versus the number of times the Update_..() functions are called, in Hp.



a) Before running the algorithm.



b) After running the algorithm.

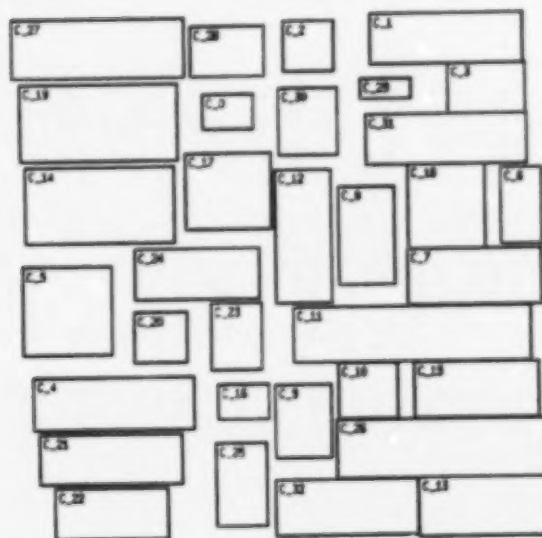
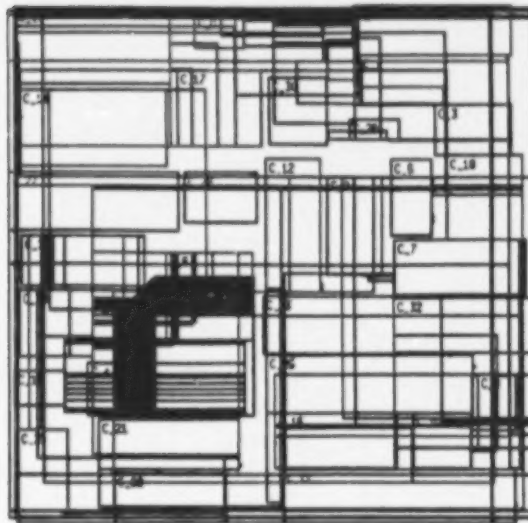


Figure A-8: Ami33 module locations produced by the best first algorithm with $k=5$.

a) Before running the algorithm.



b) After running the algorithm.

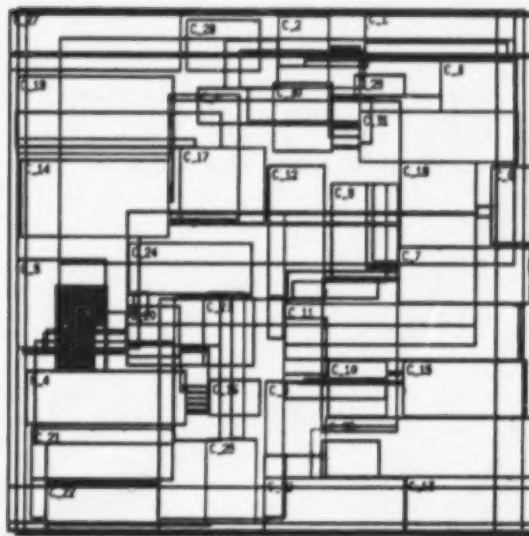


Figure A-9: Ami33 module locations and net bounding boxes, produced by the best first algorithm with $k=5$.

Figure A-10: Ami49 module locations before running the algorithm.

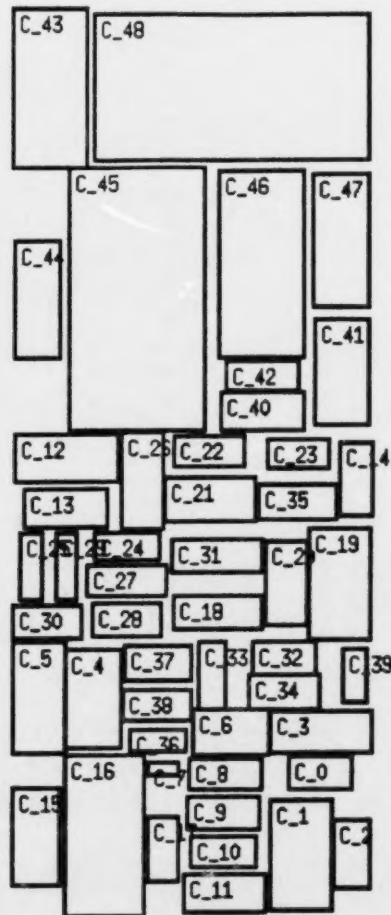


Figure A-11: Ami49 module locations after running the best first algorithm with k=5.

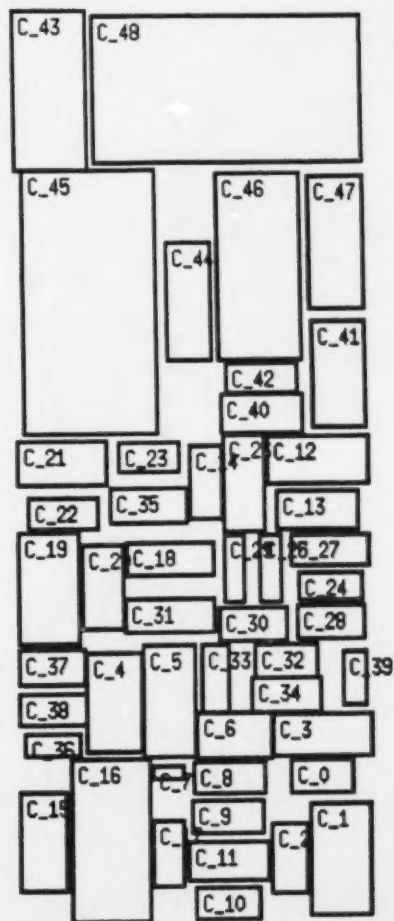


Figure A-12: Ami49 module locations and net bounding boxes before running the algorithm.

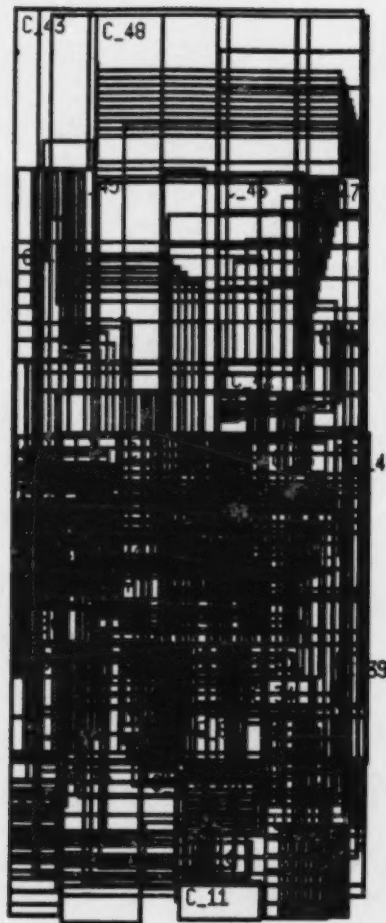


**Figure A-13: Ami49 module locations and net bounding boxes
after running the best first algorithm with k=5.**

